Pascal–2 V2.1/RT-11 Utilities Guide

Introduction to the Utilities Guide

The Pascal–2 utilities are a collection of programs designed to make life easier for programmers. Some of the utilities, such as the formatters, are designed to lessen the tedium in formatting programs and program documentation. Other utilities, such as the cross-reference programs, can help analyze code. Still other utilities, such as the MACRO package or the string-processing package, extend the capabilities of Pascal–2.

Each section of the Utilities Guide describes the particular utility in detail and includes examples on its use. Briefly, however, the Utilities Guide contains the following:

Two Program Formatters: PASMAT, a sophisticated formatter with a number of options; PB, a simple formatter designed to assist, rather than supplant, your own formatting of program text.

Two Cross-Reference Programs: XREF, which cross-references the variables in your program or words in a text file; and PROCREF, which cross-references the procedures in your program.

Dynamic String Package: STRING.PAS, a set of procedures designed to help you manipulate character strings.

MACRO Package: PASMAC, which helps to interface MACRO-11 routines with Pascal–2 programs.

Text Formatter: PROSE, which provides a number of formatting options for the production of computer-related documentation.

PASMAT: A Pascal–2 Formatter

PASMAT generates a standard format for Pascal code. PASMAT will accept standard Pascal and the language extensions in Pascal–2. PASMAT accepts full programs, external procedures, or groups of statements. A syntactically incorrect program will cause PASMAT to abort and to cease formatting the output file.

PASMAT's default formatting requires no control from you. The best way to find out how the formatting works is to try it and see. In addition, PASMAT's formatting directives give you considerable control over the output format when you wish.

Overview of Capabilities

PASMAT has these capabilities:

- The program may be converted to uniform case conventions, under the control of the user.
- The program is indented to show its logical structure and to fit into a specified output line length.
- Comment delimiters are changed to braces ({}).
- If requested, the break character (_) will be removed from identifiers for use at installations that do not support the break character.
- If requested, the first instance of each identifier will determine the appearance of all subsequent instances of the identifier.
- All non-printing characters are removed; this feature is useful after certain editing bugs.

PASMAT handles comments, statements, and tables in the following manner:

Comments

PASMAT's rules allow you to achieve almost any effect needed in the display of comments.

- A comment standing alone on a line will be left-justified to the current indention level, so that it will be aligned with the statements before and after it. If it is too long to fit with this alignment, it will be right-justified.
- A comment that begins a line and continues to another line will be passed to the output unaltered, indention unchanged. This type of comment is assumed to contain text formatted by the author, so it is not formatted.
- If a comment covered by one of the above rules will not fit within the defined output line length, the output line will be extended as necessary to accommodate the comment. Once formatting is complete, a message to the terminal will give the number of times the width was exceeded and the output line number of the first occurrence.
- A comment embedded within a line will be formatted with the rest of the code on that line. Breaks between words within a comment may be changed to achieve proper formatting, so nothing that has a fixed format should be used in such a comment. If a comment cannot be properly spaced so that the line will fit within the output length, that line will be extended as necessary. Once formatting is complete, a message to the terminal will give the number of times the width was exceeded and the output line number of the first occurrence. If no code follows a comment in the input line, then no code will be placed after the comment in the output line.

Statement Bunching

The normal formatting rule for a **case** statement places the selected statements on a separate line from the **case** labels. The **B** directive (see below) tells the formatter to place these statements on the same line as the **case** labels if the statements will fit.

Similarly, the rules for **if-then-else**, **for**, **while**, and **with** place the controlled statements on separate lines. The **B** directive tells the formatter to place the controlled statement on the same line as the statement header if the statement will fit.

Tables

Many Pascal programs contain lists of initialization statements or constant declarations that are logically a single action or declaration. You may want these to be fit into as few lines as possible. The **S** directive (see below) allows this. If this is used, logical tab stops are set up on the line, and successive statements or constant declarations are aligned to these tab stops instead of beginning on new lines.

At least one blank is always placed between statements or comment declarations, so if tab stops are set up at every character location, statements will be packed on a line.

Structured statements, which normally format on more than one line, are not affected by this directive.

Using PASMAT

Invoke PASMAT with the following command:

```
.R PASMAT
*output-file = input-file /options="directives"
```

input-file:
: The Pascal source file being reformatted. PASMAT accepts only one input file. The default file extension for both input and output is .PAS.

output-file:
: The reformatted Pascal source file. If *output-file* is omitted, the output file receives the same file name and extension as the input file and becomes the latest version of that file.

options="*directives***"**:
: Settings for formatting directives. The **options** switch is optional. It may be abbreviated to **o** and may be placed anywhere on the command line. Though the '=' is shown as the switch separator, a colon (:) may also be used between the **options** switch and the directives. When specified on the command line, directives must be placed in quotes as shown. The *directives* field will be scanned as though the directives were in a Pascal comment at the start of the source program.

Formatting Directives

Formatting directives may be specified either by an **options** switch on the command line or by a special form of the Pascal comment structure.

Formatting directives are of two breeds: switches that turn *on* with the plus sign (+) and *off* with the negative sign (-) (e.g., R+ and L-); or numeric directives of the form T=5. Multiple directives are separated by commas (e.g., R+,L-). Blanks are not allowed within a directive. Case is ignored: R+ is the same as r+ in a directive.

By definition (and by default), certain directives override other directives, such as the L directive overriding the U and R directives. Therefore, when turning on a directive, you must turn off any directive that overrides it. For example, suppose you want all Pascal reserved words in upper case. In addition to setting R+, which specifies upper case, you must also turn off the L directive with L-. See the second example under "PASMAT Examples."

The following example shows a program named PROG.PAS being formatted with a command-line directive that sets the switch B on, R off and the numeric directives O to 72 and T to 5.

```
.R PASMAT
*PROG/OPTIONS="B+,O=72,T=5,R-"
```

If used in the program text as part of an embedded Pascal comment, format directives are placed within square brackets that, along with any other comments, are placed within the standard Pascal comment braces. A compiler directive (e.g., **$nomain**), if present, must begin any comment containing a PASMAT directive. In this case, the PASMAT directive may come before or after any other text:

{$*compiler-directives text* [*directives*] *text*}

If no compiler directive is present, the PASMAT directive must begin the comment:

{[*directives*] *text*}

The following embedded directive has the same effect as the command-line directive shown above.

```
{[b+,o=72,t=5,r-]}
```

The PASMAT formatting directives are:

A (Default A-) Adjusts each identifier so that the first instance of the identifier determines the appearance of all subsequent instances of the identifier. This facility standardizes the use of upper-case and lower-case characters and the break character (_) in program text. This directive overrides the U directive.

B (Default B-) Specifies that the statements following a **then**, or **else**, **for**, **with** or **while** will be put on the same line if they will fit. The statement following a **case** label will be put on the same line if it fits. The result is a shorter output, which may be easier to read but which also may be harder to correct.

C (Default C-) Converts leading blanks to tabs on output.

F (Default F+) Turns formatting on and off. This directive goes into effect immediately after the comment in which it is placed and can save carefully hand-formatted portions of a program.

K (Default K-) Converts the Pascal–1 **else** clause in a **case** statement to **otherwise** as used in Pascal–2.

L (Default L+) Specifies that the case of identifiers and reserved words be a literal copy of the input. This directive overrides the U and R directives and is disabled by the P+ directive.

M (Default **M+**) Converts all alternate symbol representations to the standard form. Otherwise, all symbols are left as they are in the text. The nonstandard comment brackets **/* ... */** are always converted, either to braces or, in the case of **M-**, to **(* ... *)**.

N (Default **N-**) Inserts no new lines into the output unless they are required to make the lines fit. This directive just indents the source, keeping the line structure set up by the user. If a line exceeds the output length, it will be broken at the best place available, but the results may not be what you want. Look things over carefully after using this option.

O (Numeric directive, default **O=78**) Specifies the width of the output line. The maximum value allowed is 132 characters. If a particular token will not fit in the width specified, the line will be lengthened accordingly, and a message at the end of the formatting will give the number of times the width was exceeded and the output line number of the first occurrence.

P (Default **P-**) Sets "portability mode" formatting, which removes break characters (_) from identifiers. The first letter of each identifier, and the first letter following each break character, will be made upper case, while the remaining characters will be in lower case. This directive overrides the **L** and **U** directives. The **R** directive sets the case of reserved words.

Warning: Pascal-2 considers break characters significant: **User_DoesThis** is one identifier and **UserDoes_This** is another. Take care when using this directive that you do not make two different identifiers the same: **UserDoesThis** and **UserDoesThis**.

R (Default **R-**) Specifies that all reserved words will be in upper case. With this off, reserved words will be in lower case. The **L** directive overrides the **R** directive. When using **R+** you must also use **L-** to turn off the overriding directive. See the second example under "PASMAT Examples."

S (Numeric directive, default **S=1**) Specifies the number of statements per line. The space from the current indention level to the end of the line is divided into even pieces, and successive statements are put on the boundaries of successive pieces. A statement may take more than one piece, in which case the next statement again goes on the boundary of the next piece. This is similar to the tabbing of a typewriter.

Any statement requiring more than one line will not be affected, but may cause unexpected results on following statements. This directive only affects the constant declaration and statement portions of the program and is intended for use in initializing tables. The default value of 1 provides normal formatting.

T (Numeric directive, default **T=2**) Specifies the amount to "tab" for each indention level. Statements that continue on successive lines will be additionally indented by half the value of **T**.

U (Default **U-**) **U+** specifies that identifiers are converted to upper case; **U-** specifies that they will be converted to lower case. The **L**, **P** and **A** directives override this directive. When using **U+** you must also use **L-** to turn off the **L** directive. Also, make sure the **P** directive is off (**P-**, the default).

Limitations and Errors

PASMAT is limited in these ways:

- The maximum input line length is 132 characters.
- The maximum output length is 132 characters.
- Only syntactically correct statements are formatted. A syntax error in the code will cause the formatting to abort. An error message will give the input line number on which the error is detected. The error checking is not perfect, and successful formatting is no guarantee that the program will compile.
- The number of indention levels handled by PASMAT is limited; PASMAT will abort if this number is exceeded — a rare circumstance.
- If a comment will require more than the maximum output length (132) to meet the rules given, processing will be aborted. This situation should be even rarer than indention-level problems.
- When it aborts, PASMAT attempts to copy the rest of the file. You should, however, recover a copy of the source file and inspect the PASMAT-generated copy carefully; we cannot guarantee that PASMAT will recover all the text for every error condition.

PASMAT Examples

To show how the various PASMAT options work together, we will take the sample program that follows and show how it appears after reformatting with two different sets of options.

```
program Efact(output);
{ Compute an approximation for E from its Taylor series }
{ The Nth term in the series is 1/(N!)                   }
var E, series_term: real; N: integer;
begin
{ set initial conditions }
E := 1.0; N := 1; SeriesTerm := 1.0;
{ loop to approximate E; quit when the series sum stops changing }
repeat
E := E + seriesterm;
{ compute next term of series }
N := N + 1; seriesterm := seriesterm / N;
until E = (E + SeriesTerm);
writeln('With ', n: 1, ' terms, value of e is', e: 18: 15);
end.
```

First we reformat the program using the standard indention of text and comments. We use the **output** directive on the command line to specify the width of the output line, and we specify a short line width to illustrate the right-justification of long comments.

The program is formatted with the commands:

```
.R PASMAT
*EFACT/OPTIONS="O=66"
```

Program text after formatting:

```
program Efact(output);
{ Compute an approximation for E from its Taylor series }
{ The Nth term in the series is 1/(N!)                   }

  var
    E, series_term: real;
    N: integer;

  begin
    { set initial conditions }
    E := 1.0;
    N := 1;
    SeriesTerm := 1.0;
{ loop to approximate E; quit when the series sum stops changing }
    repeat
      E := E + seriesterm;
      { compute next term of series }
      N := N + 1;
      seriesterm := seriesterm / N;
    until E = (E + SeriesTerm);
    writeln('With ', n: 1, ' terms, value of e is', e: 18: 15);
  end.
```

The second example illustrates embedded PASMAT commands. We have altered the original program by inserting the text `{[A+,L-,R+]}` before the first line. The directive **A+** changes each identifier to

match the appearance of the first use of that identifier. (Notice the variant forms of **series_term** and **E** in the original program.) The directives **L-** and **R+** together turn off the literal reproduction of the reserved words and make them upper case. The program is formatted with the commands:

```
.R PASMAT
*EFACT
```

Program text after formatting:

```
{[A+,L-,R+]}
PROGRAM Efact(output);
{ Compute an approximation for E from its Taylor series }
{ The Nth term in the series is 1/(N!)                   }

  VAR
    E, series_term: real;
    N: integer;

  BEGIN
    { set initial conditions }
    E := 1.0;
    N := 1;
    series_term := 1.0;
    { loop to approximate E; quit when the series sum stops changing }
    REPEAT
      E := E + series_term;
      { compute next term of series }
      N := N + 1;
      series_term := series_term / N;
    UNTIL E = (E + series_term);
    writeln('With ', N: 1, ' terms, value of e is', E: 18: 15);
  END.
```

PB: A Pascal–2 Formatter

PB is designed on the premise that a formatting program can't do everything, that formatting requires an understanding of the meaning of a program. Thus, PB is meant to assist, rather than replace, the manual arrangement of program format. The simple transformations performed by PB reduce the tediousness of formatting program text and help ensure consistency within a variety of personal formatting styles.

PB can be used on code as it is being developed, even code that is incomplete or incorrect. You write a program, or program fragment, to some level of detail, then run it through PB. You can then edit the resulting code to alter its meaning or improve its appearance, and use PB again. This cycle can be repeated as the code progresses from initial idea to working program, and later as the program is "maintained."

Text produced by PB usually looks much like the input. Each input line is transformed into a single output line containing essentially the same text; within the code on a line the spacing is the same; and simple statements that continue onto multiple lines stay lined up.

PB adjusts program format to be consistent with the syntactic structure of Pascal. Statements at the same nesting level line up, and indention increases with the nesting level. Where possible, trailing comments are lined up with one another. Keywords and identifiers in the text are altered to match the capitalization style of their first occurrence (which may be in an included file).

Using PB

You invoke PB with the following command:

```
.R PB
*output-file = input-files /switches
```

input-files:
: The Pascal source files being reformatted. The default file extension is .PAS. Multiple files, if specified, are separated by commas. Multiple files are concatenated to produce the output file.

output-file:
: The reformatted Pascal source file. The default file extension is .PAS. If *output-file* is not specified, the output file will be created with the same file name and extension as the last input file and becomes the latest version of the file.

switches:
: Command-line switches used to adjust PB formatting. These switches must be placed after the input file names on the command line. *Switches* may be either or both of these:

  The `indent:`*num* switch specifies the number of columns (*num*) in an indention step (the space that text is shifted when the nesting level changes). The default setting is 2; larger values make the separation between levels clearer, but may force text past the right margin.

  The `comment:`*num* switch specifies the column (*num*) to which trailing comments are indented. The default setting is 33; this value works well when trailing comments are used primarily to annotate declarations.

  The switches may be abbreviated to two letters. Multiple switches are separated by a slash '/'.

Example

This example illustrates the functions provided by PB. The example also shows a particular development style, discussed above, to which PB is suited.

We start with the program at an intermediate stage in its development. Some new code has just been added. Notice that the new code is not indented; it is broken into reasonable lines, but we will let PB do the rest of the formatting.

```
var I, X: integer;
begin
  X := 1;
  for I := 1 to n do begin
repeat
x := x + 1;
prim := x is a prime number;
until prim;
write(X);
  end;
end.
```

Processing by PB gives this result:

```
var I, X: integer;
begin
  X := 1;
  for I := 1 to n do begin
    repeat
      X := X + 1;
      prim := X is a prime number;
    until prim;
    write(X);
    end;
end.
```

The indent changes at most one step from line to line. When control constructs appear one per line (the **repeat** statement, for instance) each causes the indent to increase by one step, but when a second one appears on a line it has no effect on indention. In the example, this has been used to cut the "noise" from **begin**...**end** brackets.

Notice a couple of things at this stage. First, PB does not know much about Pascal language syntax (note the phrase "**X is a prime number**"). Second, PB uses the first instance of a word, regardless of context, for its capitalization style (you can set the style for an identifier by adjusting its declaration, since the declaration of an identifier must precede its use).

Of course, the program is not yet complete, nor does it contain any comments. The following is closer to a final version.

```
type Index = 1..n;
var
X: integer; { number being tested for primality }
j, { count of primes found }
k, { trial divisor }
lim: index; { last divisor to test }
Prim: boolean; { 'true' until a divisor is found }
P: array[index] of integer; { P[I] = Ith prime number }
begin
  P[1] := 2; X := 1; Lim := 1;
write('2');
  for J := 2 to n do begin
    repeat
      X := X + 2;
if sqr(P[Lim]) <= X then Lim := Lim + 1;
K := 2; Prim := true;
while Prim and (K < Lim) do begin
Prim := (X mod P[K]) <> 0;
K := K + 1;
end;
    until Prim;
    P[J] := X; write(X);
    end;
end.
```

Processing by PB gives:

```
type Index = 1..n;
var
  X: integer;                    { number being tested for primality }
  j,                             { count of primes found }
  k,                             { trial divisor }
  lim: Index;                    { last divisor to test }
  Prim: boolean;                 { 'true' until a divisor is found }
  P: array[Index] of integer;    { P[I] = Ith prime number }
begin
  P[1] := 2; X := 1; lim := 1;
  write('2');
  for j := 2 to n do begin
    repeat
      X := X + 2;
      if sqr(P[lim]) <= X then lim := lim + 1;
      k := 2; Prim := true;
      while Prim and (k < lim) do begin
        Prim := (X mod P[k]) <> 0;
        k := k + 1;
        end;
    until Prim;
    P[j] := X; write(X);
    end;
end.
```

The comments have been moved to the right, where they stand apart from the program code and line up for easier reading. If we had wanted to keep a comment attached to the code, we could have placed it in front of the final comma or semicolon on its line (then it would not be a trailing comment and would be treated as part of the text), or placed it on a line by itself (where it would be aligned at the prevailing indent).

Detailed Formatting Rules

Indention is directed by the nesting of control constructs in the program text. Generally, when the nesting level increases, the indent increases by the indention step; when a nesting level ends, the indent reverts to that of the surrounding nesting level. A change of indention at the beginning of a line takes effect immediately; otherwise, it takes effect on the next line. The exceptions to the rules are:

- The start of a new nesting level does not change the indent if it begins on the same line as the surrounding level.
- When a line begins with a statement label, the indent for that line is decreased by the indention step.

Normal indention rules do not apply when a simple statement or clause continues across multiple lines. In these cases, the initial line is indented normally, but the following lines are arranged to preserve their alignment with the initial line. Changes of indention within continued lines take effect after the last continuation.

A similar adjustment occurs when a comment continues across multiple lines. When a trailing comment is continued, the following lines stay aligned with the initial part of the comment, but not necessarily with the rest of that line (a trailing comment may shift in relation to the rest of the line).

The following constructs affect the indention level:

- A *program-declaration*, *procedure-declaration* or *function-declaration* is arranged so that the *heading*, the keyword introducing a *label-declaration-part*, *const-definition-part*, *type-definition-part*, or *variable-declaration-part*, and the *body* are all at the same indention level. The list of declarations within a *label-declaration-part*, *const-definition-part*, *type-definition-part*, *variable-declaration-part*, or *procedure-and-function-declaration-part* is set one indention step deeper.
- The *component-type* of an *array-type* or *file-type*, and the *base-type* of a *set-type* are indented one more step. Within a *record-type* the *field-list* is indented another step, the list of *variants* within the *variant-part* is indented an additional step, and the *field-lists* within individual *variants* are indented yet another step.
- The *statement-sequence* within a *compound-statement* is indented an additional step.
- The controlled *statement* within a *for-statement*, *if-statement*, *else-part*, *while-statement* or *with-statement*, and the *statement-sequence* within a *repeat-statement* are indented another step. Within a *case-statement* the list of *case-list-elements* is indented one more step, and the controlled *statement* of each *case-list-element* is indented an additional step.

XREF: A Pascal–2 Cross-Reference Lister

XREF produces a cross-reference listing of the identifiers in a Pascal program. XREF is helpful when debugging new programs or when modifying existing ones. The output shows the use of each identifier in the program, which is beneficial when you're working with medium to large programs.

Each identifier is listed, along with an entry for each reference to that identifier. Each entry consists of the line on which the reference occurs, plus an indication of whether the reference is a declaration or assignment.

Using XREF

You invoke XREF with the following command:

```
.R XREF
* output-file = input-file /switches
```

input-file:
: The Pascal source file being cross-referenced. The input file has a default extension of .PAS. XREF accepts only one input file.

output-file:
: The cross-reference file. The output file has a default extension of .CRF. *Output-file* and the '=' separator are optional. If they are omitted, an output file with the same name as the input file, and having the default extension, is placed in the default directory.

switches:
: Command-line switches. *Switches* may be either or both of these:

  The `list` switch generates a listing of the input file before the cross-reference. This listing includes line numbers and a flag character (c) indicating multiple line comments and strings. The flag character makes it easier to locate certain bugs that cannot be easily diagnosed by the compiler.

  The `width:`*num* switch specifies the page width for the cross-reference listing, where *num* is the number of characters across. The default is 132.

  The switches may be abbreviated to one letter. Multiple switches are separated by a slash '/'.

Limitations

The XREF program has two limitations on the size of the programs it can handle.

- An internal limit exists for the number of distinct identifiers allowed. You can change this number in the XREF source file and recompile the program.
- The total number of references is limited by the amount of dynamic storage available.

The XREF program does not perform a complete syntax analysis of the program, and it may not flag all declarations or assignments.

Example of XREF Listing

This example shows the cross-referencing of the program EFACT to produce the output file EFACT.CRF.

```
.R XREF
*EFACT/LIST/WIDTH:66

      1 program Efact(output);
      2 { Compute an approximation for E from its Taylor series.
c     3    The Nth term in the series is 1/(N!).
c     4 }
      5
      6   var
      7     E, SeriesTerm: real;
      8     N: integer;
      9
     10   begin
     11     { set initial conditions }
     12     E := 1.0;
     13     N := 1;
     14     SeriesTerm := 1.0;
     15     repeat  { loop to approximate E; quit when sum stops changing }
     16       E := E + SeriesTerm;
     17       N := N + 1;
     18       SeriesTerm := SeriesTerm / N;
     19     until E = (E + SeriesTerm);
     20     writeln('With ', N: 1, ' terms, value of e is', E: 18: 15);
     21   end.


Cross reference:  * indicates definition, = indicates assignment

-E-
E                 7*    12=    16=    16     19     19     20
EFACT             1*

-I-
INTEGER           8

-N-
N                 8*    13=    17=    17     18     20

-O-
OUTPUT            1*

-R-
REAL              7

-S-
SERIESTERM        7*    14=    16     18=    18     19

-W-
WRITELN          20

end xref   8 identifiers    24 total references
```

PROCREF: Pascal–2 Procedural Cross-Reference Lister

PROCREF, based on a procedural cross-reference program published by Arthur Sale in *Pascal News* (Number 17, March 1980), is designed to help programmers sort through the procedures in medium to large Pascal programs. The program has been modified to allow the use of multiple input files and **%include** directives and to provide "called by" data in the listing.

PROCREF provides a quick overview of the procedural organization of a program, which is beneficial when you're working with medium to large programs. The PROCREF utility reads the text of a Pascal program to produce a compact listing of the procedure headings and an alphabetized list of procedures with usage information. PROCREF processes **%include** directives in the same way as the Pascal–2 compiler, so that all parts of a compilation can be analyzed.

The procedure listing includes each procedure heading, along with its location in the input file. Procedure headings are indented to show lexical level. No attempt is made to fit the procedure headings into a limited line width.

The cross-reference listing places procedures in alphabetical order. For each procedure the listing includes:

- The file and line number where its heading starts.
- The file and line number where its body starts, unless it is external or is a formal procedure parameter and has no body. In such a case, the note **external** or **formal** is printed.
- If the procedure was declared **forward** or is externally defined, the listing contains the file and line number where the procedure heading stub starts.
- A list of all procedures immediately called by this procedure. These are listed in the order in which they occur in the text. A procedure is listed only once, even if it is called more than once.
- A list of all procedures that call this procedure. Again, the list is in textual order and only one reference is shown per procedure.

Only the first sixteen characters of a procedure name appear in the cross-reference listing. Those characters are written exactly as they appear in the program text.

Using PROCREF

You invoke PROCREF with the following command:

```
.R PROCREF
* output-file = input-files /width:num
```

input-files:
: The Pascal source files being cross-referenced. The input files have a default extension of .PAS. Multiple input files, if specified, are separated by commas. Multiple files will be concatenated.

output-file:
: The cross-reference file. The output file has a default extension of .PRF. The *output-file* and the '=' separator are optional. If they are omitted, an output file with the same name as the last input file, and having the default extension, is placed in the default directory.

width:*num*
: Specifies the page width for the cross-reference listing, where *num* is the number of characters across the page. The default is 80 characters. The **width** switch is optional and may be abbreviated to one letter.

Note that the RT-11 system will automatically truncate the name PROCREF to PROCRE.

Limitations

The PROCREF program does not do a complete syntax analysis of the program being processed. PROCREF will err in one case: If a field identifier in a record has the same name as a procedure, and if that field is referenced without a preceding record variable name, as in a **with** statement, the field identifier will be treated as a reference to the procedure.

Example

Let's assume that we wish to generate a procedure cross-reference for the following program, LVSPOOL.PAS.

```
Pascal-2 RT11 SJ V2.1A   5-Aug-83   7:04 PM      Site #1-1      Page 1-1
Oregon Software, 2340 SW Canyon Road, Portland, Oregon 97201, (503) 226-7760
LVSPOOL/LIST

    1       program LVSpool(input, output);
    2         procedure ScanLV; external;
    3         procedure ReadFontInfo(i: integer; j: integer); forward;
    4
    5         procedure LoadFonts;
    6           procedure GetByte;
    7           begin
    8           end;
    9         begin
   10           GetByte;
   11           ReadFontInfo(1, 2);
   12         end;
   13
   14         procedure ReadFontInfo;
   15         begin
   16           LoadFonts;
   17         end;
   18
   19         procedure ShowPage;
   20         begin
   21           ScanLV;
   22         end;
   23
   24       begin           main program
   25         ReadFontInfo(0,1);
   26         ShowPage;
   27       end.


*** No lines with errors detected ***
```

We cross-reference the procedures as follows.

```
.R PROCREF
*LVSPOOL=LVSPOOL/W:72
```

The W:72 requests that the cross-reference listing not exceed 72 characters in width so that the result may be printed on a terminal. The result, placed in the file LVSPOOL.PRF, consists of:

```
Procedural Cross-Referencer - Version 3.0
LVSPOOL/W:72

 Line Program/procedure/function heading
-----------------------------------------

LVSPOOL.PAS:

    1  program LVSpool(input, output);
    2      procedure ScanLV; external;
    3      procedure ReadFontInfo(i: integer; j: integer); forward;
    5      procedure LoadFonts;
    6          procedure GetByte;
   14      procedure ReadFontInfo;
   19      procedure ShowPage;

Procedural Cross-Referencer - Version 3.0
LVSPOOL/W:72

Cross Reference Listing

GetByte             Head: LVSPOOL.PAS, 6  Body: LVSPOOL.PAS, 7
      Called by     LoadFonts

LoadFonts           Head: LVSPOOL.PAS, 5  Body: LVSPOOL.PAS, 9
      Calls         GetByte          ReadFontInfo
      Called by     ReadFontInfo

LVSpool             Head: LVSPOOL.PAS, 1  Body: LVSPOOL.PAS, 24
      Calls         ReadFontInfo     ShowPage

ReadFontInfo        Head: LVSPOOL.PAS, 3  Body: LVSPOOL.PAS, 15
                    Forward, header stub: LVSPOOL.PAS, 14
      Calls         LoadFonts
      Called by     LoadFonts        LVSpool

ScanLV              Head: LVSPOOL.PAS, 2  external
      Called by     ShowPage

ShowPage            Head: LVSPOOL.PAS, 19  Body: LVSPOOL.PAS, 20
      Calls         ScanLV
      Called by     LVSpool
```

Dynamic String Package

The Pascal standard implements character strings in two ways: as a sequence of two or more characters between single-quote marks (a literal string); or as a **packed array of char** (a variable string). However, the standard does not provide adequate facilities for manipulating character strings and only allows assignments of one string to another string and comparisons of two strings of equal length.

Pascal–2's Dynamic String Package extends the meager string-handling capabilities of standard Pascal, providing the ability to perform sophisticated operations on strings of varying lengths. The string package, STRING, is a collection of string-processing procedures and functions that allows Pascal programs to read and write strings, concatenate two strings, search one string for another, insert one string into another and delete one string from another, assign the value of one string to another string and other string operations. The string package is written in standard Pascal to take advantage of conformant array parameters, which facilitate the passing of variable-length arrays (strings), and to provide portability to other Pascal implementations.

To use the string package, declare string variables as packed arrays of characters with a lower bound of 0 and an upper bound equal to the maximum length for that particular string, as shown:

```
var
  string-name: packed array [0..max-len] of char;
```

where *string-name* is the identifier associated with the string variable and *max-len* is the maximum length of the string in bytes. The actual length of the string is stored in element 0. The characters making up the string are stored starting at element 1. *Max-len* must be greater than 0 and no larger than 255.

The maximum length of a string may be different for each string, depending on the intended use of the string. The string package's use of conformant array parameters makes this possible. Examples:

```
var
  NameString: packed array [0..25] of char;
  SiteNo: packed array [0..7] of char;
  LineOfInput: packed array [0..80] of char;
```

As an alternative, these routines also accept parameters of type **packed array** [1..*max-len*] **of char**, where *max-len* is the actual length of the string. Literal strings are of this type. This means you may pass a literal string to any of these procedures as long as the formal parameter is not a **var** parameter.

STRING may be included in program source files in one of two ways: in the program code, place the **%include** compiler directive; or on the command line, concatenate STRING.PAS with the rest of the source files making up the program ("source concatenation"). We recommend the use of the **%include** directive (e.g., **%include 'string';**). However, if you concatenate the string package with the source file, the command to compile program PROG is:

```
.R PASCAL
*STRING,PROG
```

Source concatenation may be used only if the main program does not contain a **program** statement; otherwise, compilation errors result. Refer to "Multiple Source Files" in the Programmer Reference for more information on the **%include** directive.

The Procedures and Functions

In the definitions below, *string* and *target* represent string variables similar to the previous examples. *File* must be a variable of type **text**. *Start* and *span*, of type **integer**, represent character positions and character ranges, respectively. *Max-len* is the upper boundary, or maximum length, of the array. *Char* may be a variable of type **char** or a literal string of one character.

The string package contains these procedures and functions:

Len(*string*)
: An integer function, returns the actual length of *string*. *String* may be a literal string.

Clear(*string*)
: Initializes *string* to empty.

ReadString(*file*, *string*)
: Reads *string* from *file*. The string is terminated when **eoln**(*file*) becomes **true**, and a **readln**(*file*) is performed. Overflow results in truncation to *max-len* characters.

WriteString(*file*, *string*)
: Writes *string* to *file*. This procedure does not accept literal strings as parameters. Use **writeln** to terminate a written string manually.

Concatenate(*target*, *string*)
: Appends *string* to *target*. The resulting value is *target*. *String* may be a literal string. Overflow results in truncation to *max-len* characters.

Search(*string*, *target*, *start*)
: Searches *string* for the first occurrence of *target* to the right of position *start* (characters are numbered beginning with 1). The **Search** function returns the position of the first character in the matching substring, or the value zero if *target* does not appear in *string*. *String* and *target* may be literal strings.

Insert(*target*, *string*, *start*)
: Inserts *string* into *target* at position *start*. Characters are shifted to the right as necessary. Overflow produces a truncated *target* of *max-len* characters. The insertion is skipped if the *start* position causes a non-contiguous string. *String* may be a literal string.

Assign(*target*, *string*)
: Assigns *string* to *target*. This procedure is especially useful for assigning a literal *string* to a variable string (*target*). To assign one character to a variable string, use the **Asschar** procedure, below.

Asschar(*target*, *char*)
: Assigns *char* to *target*. *Char* may be a literal character or a variable name. This procedure is more efficient than procedure **Assign** for the creation of one-character strings. (With **Assign**, a one-character string must first be created as input to **Assign**, which then assigns the character to a variable string.)

Equal(*target*, *string*)
: Determines whether *target* is element-for-element identical to *string*. This boolean function returns a **true** value if the two strings are equal, **false** if the two strings are different. *Target* and *string* may be literal strings.

The *start* and *span* parameters in the **Delstring** and **Substring** procedures define a substring beginning at position *start* (between characters *start*-1 and *start*) with a length of **abs**(*span*). If

span is positive, the substring is to the right of *start*; if negative, the substring is to the left.

Delstring(*string, start, span*)
Deletes the substring defined by *start, span* from *string*. (In previous versions of Pascal-2, this procedure was named **delete**.)

Substring(*target, string, start, span*)
The substring of *string* defined by *start, span* is assigned to *target*. *String* may be a literal string.

Example

The sample program PDLIST.PAS demonstrates the use of the string package. The program reads a PROSE input file (.PRS extension) of text-processing commands, or "directives," searching for all directives used in the file. (PROSE is described later in this guide.) As each directive is encountered, PDLIST prints the directive and its location within the file for future reference.

The first character of a PROSE directive is called the "escape" character. If the first character of a line of input is an escape character, at least one directive follows. PDLIST.PAS uses the **Readstring** procedure to read a line of text as a string, then calls **Search** repeatedly to find each occurrence of the escape character on the current line. When an escape character is found, the procedure **GetDirective** is called to get the next directive. For each directive, the program builds a line of output (also a string) using the **Assign** and **Concatenate** procedures, and uses **Writestring** to write the line to a .DTV (directive) file. In this example, the escape character is a period, which is the PROSE default.

PDLIST.PAS, including procedure **GetDirective**, is provided in full on the following pages. Sample execution follows the listing.

```
program DirectiveList;
  %include 'string';  ————————————————— include the string package

  const
    LineLength = 150;    { PROSE default input line length }

  var
    Line: packed array [0..LineLength] of char;   { string for output line }
    Outline: packed array [0..50] of char;        { string for input line }
    Directive: packed array [0..10] of char;      { string for directive }
    Name: packed array [1..80] of char;           { input file name }
    Escape: packed array [0..1] of char;          { string for escape character }
    Linenum, Index: integer;
    Prosefile, Directivefile: text;
    DirectiveFound: boolean;          { directive found? }
    NoneYet: boolean;                 { looking for first directive on line }
    Letters: set of char;             { Characters making up a directive }
```

```
procedure GetDirective(I: integer);
  var
    Ch: char;

  begin   { GetDirective }
    Clear(Directive);  ———————————————— defined in STRING
    while I <= Len(Line) do begin    { not end of line yet }
      Ch := Line[I];  I := I + 1;
      if Ch in Letters then begin     { get next char of directive }
        Directive[0] := succ(Directive[0]);
        Directive[Len(Directive)] := Ch
        end
      else I := Len(Line) + 1;
      end;
    DirectiveFound := Len(Directive) > 0
  end;     { end GetDirective }

  begin   { DirectiveList }
    write('PROSE FILE: ');
    readln(Name);
    reset(Prosefile, Name, '.prs');
    rewrite(Directivefile, '.dtv', Name);
    Assignchar(Escape, '.');  ———————————— defined in STRING
    Linenum := 0;
    Letters := ['A'..'Z', 'a'..'z'];      { Any others end directive }
    while not eof(Prosefile) do begin
      ReadString(Prosefile, Line);  ———————————— defined in STRING
      Linenum := Linenum + 1;
      if Line[1] = Escape[1] then begin   { first character is an escape }
        Index := 0;  NoneYet := true;
        repeat      { find all occurrences of escape characters }
          Index := search(Line, Escape, Index + 1);   — defined in STRING
          if Index <> 0 then begin
            GetDirective(Index + 1);      { get the next directive }
            if DirectiveFound then begin
              Assign(Outline, Escape);  ———————— defined in STRING
              Concatenate(Outline, Directive);  ——— defined in STRING
              if NoneYet then begin
                write(Directivefile, ' Line ', Linenum: 4,'    ');
                NoneYet := false
                end
              else write(Directivefile, '               ');
              WriteString(Directivefile, Outline);   — defined in STRING
              writeln(Directivefile);
              end;
          end;
        until Index = 0;
        end;
    end;
  end.  { DirectiveList }
```

For this illustration, PDLIST reads the PROSE input file presented in Appendix A of "PROSE: A Text Formatter," Example 2, later in this guide. The name of the input file is PEXAM2.PRS. To

see what the input looks like, refer to the aforementioned example in the PROSE section.

Run PDLIST using these commands:

```
.R PASCAL
*PDLIST

.LINK PDLIST,SY:PASCAL
.RUN PDLIST
PROSE FILE: PEXAM2
```

Output — the list of directives used in the file — is written to PEXAM2.DTV, which looks like this:

```
Line     1     .COMMENT
Line     2     .INPUT
Line     3     .OPTION
Line     4     .FORM
Line     5     .MARGIN
Line     6     .PARAGRAPH
Line    21     .OPT
Line    22     .MAR
Line    23     .PARAGRAPH
Line    28     .OPT
               .MAR
               .PAR
```

MACRO-11 Procedures With Pascal–2

Although most programs can be written within the Pascal–2 language, applications involving interface to the operating system require the use of MACRO-11 assembly language code. A set of macros provided with the Pascal–2 system makes this interface easy. You can code a set of macro calls that look much like a Pascal procedure declaration, and the PASMAC macro package will assign addresses to the parameters and generate procedure entry and exit code.

Design of MACRO-11 Procedures

Follow these general rules in deciding what to put in a MACRO-11 procedure:

- Do the absolute minimum in MACRO-11. If you must use MACRO-11 code to use a system service, process the result in Pascal–2 code. (This is not always possible, since some operating systems require very low-level manipulations.)
- Isolate a common function and make the procedure handle the most general case of that function.
- Pass all data to and from the procedure as parameters. Global references from MACRO-11 are not recommended for these reasons: the address is hard to find; if the Pascal program changes, the MACRO program will have to be changed; and global references cannot be checked for type compatibility. This guide does not describe ways to make global references.

Once you have decided on the contents of the procedure, define the calling sequence as a Pascal external procedure. Then write a functional description of the procedure. Then actually write the procedure. These documents will be your implementation guide.

When you have the external definition, use the PASMAC macro package described below to define parameters and local variables. As long as the stack is not changed within the procedure, these macros can access parameters or local variables directly. For this reason, you should probably store local temporary values in the local variables rather than pushing them on the stack. If thoroughly familiar with writing MACRO-11 code, you can use the stack, but make sure you understand the Pascal–2 run-time structure, described in the Programmer's Guide.

The PASMAC Macro Package

The PASMAC macro package is provided to simplify the writing of MACRO-11 procedures to interface with Pascal–2. Using this package, you can declare procedures, parameters, and variables, and you can easily refer to these items within the procedure.

The package consists of the following macros:

Name	Arguments	Function
`proc`	*procname*	Begin the declaration for the procedure *procname*.
`func`	*funcname* *result* *restype*	Begin the declaration for the function *function*. The returned value will be that assigned to *result*, of type *restype*.
`param`	*parmname* *parmtype*	Declare a parameter named *parmname* of type *parmtype*.
`var`	*varname* *vartype*	Declare a local variable named *varname* of type *vartype*.
`save`	<*reg0*, ... ,*regn*>	Specify general registers to save on procedure entry.
`rsave`	<*ac0*, ... ,*acn*>	Specify floating accumulators to save on procedure entry.
`begin`		Begin the actual procedure code. This macro generates code to push the variables on the stack and to save registers.
`endpr`		End the code for this procedure, restore registers, pop variables and parameters from the stack, and return to the calling location.

The following example demonstrates how these macros may be used in a procedure definition. Note the correspondence between the Pascal–2 code and the MACRO-11 code.

Pascal–2 procedure definition:

```
procedure Exampl(Inp1: integer;      { first value parameter }
                 Inp2: real;         { second value parameter }
                 var Outp: integer   { variable parameter });

var
  Var1: integer;                     { first local variable }
  Arr1: array [1..3] of integer;     { second local var }

begin                                { begin body of procedure }
  :  ------------------------------------------------ procedure code
end;                                 { end of procedure }
```

The corresponding MACRO-11 code:

```
proc    exampl              ; declare the procedure
param   inp1,integer        ; first value parameter
param   inp2,real           ; second value parameter
param   outp,address        ; variable parameter

var     var1,integer;       ; first local variable
var     arr1,3*integer      ; second local variable

save    <r0,r1>             ; registers being used
rsave   <ac0,ac1>           ; floating accum being used

begin                       ; begin body of code
  :   ------------------------------------------------ procedure code
endpr                       ; reset everything and return
```

Using PASMAC

The macros described in the following sections are included in the file PASMAC.MAC, which also includes definitions of standard data types. It is assumed that this file will be assembled as a header to any MACRO-11 code. This would normally be done with a command line similar to:

```
.R MACRO
*EXTPRO,EXTPRO=SY:PASMAC,EXTPRO
```

The result of this assembly is an object file (.OBJ) that is linked in the same way as any other external module.

MACRO-11 modules assemble with the PASMAC package are referenced from Pascal via the **external** directive instead of the **nonpascal** directive, because PASMAC simulates the Pascal calling sequence. (MACRO-11 routines assembled without the PASMAC package can be referenced via the **nonpascal** directive.) For example:

```
procedure ExtProc(Parm1: integer);
  external;
```

For details, see "External Modules" in the Programmer's Guide.

The example command line above also generates a listing file (.LST). Listing of the PASMAC file is disabled with a .NLIST directive at the start of the file. A compensating .LIST directive is placed at the end of the file, so a program listing is not affected. Defining the tag $LIST anywhere in your code will enable listing of the PASMAC file.

The macros depend on the existence of a uniform radix throughout the declaration of a single procedure. This radix may be octal or decimal, but it must not be changed within a procedure declaration. Also, the macros use labels of the form Q$*xxx* and macros of the form $P*xxx* for storing state data. Avoid such forms in your own code.

Procedure Definition Macros

The PASMAC procedure definition macros must be used in the order:

Macro	Usage
`proc/func`	Exactly one of these is required
`param`	As many as required (or none)
`var`	As many as required (or none)
`save/rsave`	Either or both as needed
`begin`	Required
⋮	User code
`endpr`	Required

A MACRO-11 error is detected if the macro calls are not made in the required order.

Above references to parameter and variable "types" assume that "type" identifiers are equivalent to the length of a value of that type. For example, the identifier **integer** has the value of 2, the identifier **real** has the value of 4, and a disk buffer may have the value of 512. The PASMAC package defines some standard types. See "Type Definitions" below.

Parameter, variable and function result names are set to offsets relative to the value of the stack pointer at the end of the **begin** macro. This takes into account local variables allocated on the stack, plus the space used for register saving. You must take into account any additional values you pushonto the stack.

Examples:

```
param   param1,integer     ; defines param1
  ⋮
mov     param1(sp),r0       ; use param1
```

The 'Proc' Macro

The **proc** macro, used to begin the definition of a procedure, specifies the name to be used and initializes the symbols that store data about the procedure. This macro must be the first macro used in a procedure declaration.

The calling sequence is:

```
proc    procname[, check=1]
```

where

procname
: is the name to be used to call the procedure. Only the first six characters of this name are significant.

check
: is an optional parameter specifying stack overflow checking. A non-zero value (default) requests a stack overflow check. This check is free (and always done) if more than three registers are saved, and costs two words in the procedure entry otherwise. The time for the check is very small, so disabling it is not recommended.

Examples:

```
proc    p,check=0
  or:
proc    p,0
```

Begins the declaration of a procedure with the external name **p** and no stack overflow checking.

```
proc    savetime
```

Begins the declaration of a procedure with the external name **saveti** and stack checking enabled.

The 'Func' Macro

The **func** macro, similar in function to the **proc** macro, also allows you to specify a name and type for the returned value. In Pascal, the returned value is specified by assignment to the function name. In MACRO-11, this assignment is not possible, since the function name is used for the procedure entry and cannot also point to the appropriate place on the stack. Any value assigned to the result name defined in the **func** macro at exit from the function is returned as the function value.

The calling sequence is:

```
func    funcname, resname, restype[, check=1]
```

where

funcname is the name to be used to call the function. Only the first six characters are significant.

resname is the name to be used to reference the returned value. Any value assigned to this location during execution is returned to the calling program upon exit from the procedure.

restype is the length of the result value. This is not used in the current implementation of the macros, but is included for documentation and possible future use.

check is an optional parameter that enables stack checking if non-zero. See the description under the **proc** macro.

Example:

```
func    curtime,tval,real
```

Begins the declaration of a function with the external name **curtim** and stack overflow checking enabled. The result location will be named **tval**, of type real. Here **real** is assumed to have the value 4, which is the length of a single-precision real value.

The 'Param' Macro

The **param** macro specifies parameters to the current procedure or function. Each parameter has one **param** macro, in the order declared in the Pascal procedure declaration. In the Pascal–2 calling sequence, parameters are pushed onto the stack in the order in which they are declared, so the first parameter is at a higher address than the last parameter. Value parameters have the actual value pushed, and variable parameters have the address of the variable pushed. When these parameters are declared, the parameter name is set equal to the offset of that parameter relative to the stack pointer (**sp**) after the **begin** macro has been called. This value may be used to access the parameter location relative to the stack pointer.

The calling sequence is:

```
param   paramname, paramtype
```

where

paramname
is the name to be used for accessing the parameter. Within the body of the procedure, if the stack pointer (**sp**) has not changed since the **begin** macro, value parameters can be referred to by *paramname*(**sp**), and variable parameters can be referred to as @*paramname*(**sp**).

paramtype
is the data type used to determine the space on the stack used by this parameter.

Examples:

```
param   input,integer      ; input: integer
param   result,address     ; var result: integer
```

These macros define two parameters. The first is a value parameter with the name **input** of type **integer** and is referred to in the body of the procedure as **input(sp)**. The second is a variable parameter with the name **result** of type **integer**. Note that the type is defined only in the comment; the actual value pushed on the stack is of type **address**. Within the body of the procedure this is **@result(sp)**.

The 'Var' Macro

The **var** macro, similar to the **param** macro, defines a local variable to be allocated on the stack upon procedure entry. The space for these variables is allocated automatically by the **begin** macro, but is not initialized. Such variables are referenced relative to the stack pointer (**sp**).

The calling sequence is:

```
var     varname, vartype
```

where

varname is the name to be used for accessing the variable. Within the body of the procedure, if the stack pointer (**sp**) has not been modified since the **begin** macro, variables can be referred to by *varname*(**sp**).

vartype is the data type used to determine the space to be allocated for this variable.

Example:

```
var     temp,integer      ; temp: integer;
var     name,10*char      ; name: array [1..10] of char;
```

The example defines two local variables. The space for these variables will be pushed onto the stack by the **begin** macro. The variable **temp** has two bytes allocated and is referred to as **temp(sp)**. The variable **name** has ten bytes allocated and is referred to as **name(sp)**.

The 'Save' Macro

The **save** macro specifies the general registers to be saved on procedure entry. The Pascal–2 calling conventions require a procedure to save and restore all registers used within a procedure, so any registers altered within the procedure should be listed here. If more than three registers are to be saved, a routine from the Pascal support library is used to save the registers. The stack pointer and program counter (**sp** and **pc**) cannot be saved.

The calling sequence is:

```
save    <reg1, ..., regn>
```

where <*reg1, ..., regn*> is a list of registers to be saved, enclosed in angle brackets (<>) and separated by commas. These registers will be saved on entry and restored on exit. The registers **sp** and **pc** cannot be saved, as they are modified by the action of saving them.

Examples:

```
save    <r0,r1>
```

Save registers **R0** and **R1** and restore them on exit. The code generated uses explicit **mov** instructions to do this.

```
save    <r0,r1,r2,r3,r4,r5>
```

Save and restore all available registers. Support routines will be used.

The 'Rsave' Macro

The **rsave** macro is useful only for machines with the Floating Point Processor (FPP) hardware option and serves the same function as **save** except for the floating-point accumulators. You are required to specify the FPP mode, either single or double (default is single). Since the accumulators **AC4** and **AC5** cannot be moved directly to memory, they may not be used unless one of the accumulators **AC0** to **AC3** is also used. Of course, you cannot get data into **AC4** or **AC5** without using one of the lower accumulators, so you should not have any problems meeting this requirement.

The calling sequence is:

```
rsave   <accum1, ..., accumn>[, double=0]
```

where

<accum1, ..., accumn>
: is a list of accumulators to be saved, enclosed in angle brackets (<>) and separated by commas. These registers will be saved on procedure entry and restored on procedure exit.

double
: is an optional parameter that specifies the saving of two-word accumulators. If set to 1, specifies that the FPP is in double mode. The default is zero. The setting does not affect the setting of the FPP; it simply allows the correct computation of the space required for the registers.

Examples:

```
rsave   <ac0,ac4>
```

Save accumulators **AC0** and **AC4** and assume that the FPP is in single mode.

```
rsave   <ac0>,double=1
  or:
rsave   ac0,1
```

Save accumulator **AC0** and assume that the FPP is in double mode.

The 'Begin' Macro

The **begin** macro marks the start of the procedure body. This and the **endpr** macro are the only ones to actually generate code. When the **begin** macro is assembled, all of the data saved up by the previous macros is used to generate procedure entry code and define all of the parameter and variable addresses.

The calling sequence is:

```
begin
```

The 'Endpr' Macro

The **endpr** macro marks the end of the procedure body. Only one **endpr** is allowed in each procedure. When the **endpr** is assembled, registers are restored, the variables and arguments are popped off the stack, and control is returned to the calling procedure. The **endpr** macro is designed to generate good code for popping the stack and returning.

The calling sequence is:

```
endpr
```

Type Definitions

In addition to the procedure definition macros described above, the PASMAC package defines some standard "types" and provides a set of three macros to simplify the definition of data structures. Each type is represented by its length in bytes.

The predefined types are:

Type	Length
char	1
boolean	1
scalar	1
integer	2
pointer	2
address	2
real	4
double	8
procpar	4

The type **procpar** is actually a record definition having two fields. This type is explained below.

The structure definition package consists of three macros:

Name	Argument	Function
record	*typename*	Begins the definition of a record type *typename*. The symbol *typename* will be set to the length of the record at the end of the definition. If the data type is **procpar**, the record is a procedure being passed as a parameter to a MACRO-11 routine.
field	*name* *size*	Defines a field in the record. The fields are allocated in ascending order, and any field with a length greater than 1 is allocated on a word boundary. Fields so defined are set equal to the offset of the field relative to the beginning of the structure.
endrec		Ends the definition of a record and assigns the total length to the *typename* given in the **record** macro.

For example, consider the following Pascal record definition:

```
prec = record
  Intf1: integer;
  Intf2: integer;
  Boolf1: boolean;
  Realf1: real;
  end;
```

The equivalent code using the structure-definition macros is:

```
record  prec              ; prec = record
field   intf1,integer     ;   intf1: integer;
field   intf2,integer     ;   intf2: integer;
field   boolf1,boolean    ;   boolf1: boolean;
field   realf1,real       ;   realf1: real;
endrec                    ; end;
```

Later in the procedure, where the definition above occurs, we find:

```
var     local,prec          ; local: prec
```

And we would refer to field **intf2**, for example, as

```
mov     local+intf2(sp),r0
```

The type **procpar** is used in rare cases when you are passing a procedure as a parameter to a MACRO-11 routine. The definition is:

```
record  procpar
field   pp.proc,address     ; address of the procedure
field   pp.stat,address     ; address of the enclosing static link
endrec
```

Example

This example shows the coding of a MACRO-11 procedure for use with Pascal–2. The procedure chosen for the example is not one that would normally be coded in MACRO-11, but most such procedures are extremely dependent on the operating system. In fact, we begin with a version of the algorithm as it is coded in Pascal–2:

```
{$nomain}

procedure CountOnes(N: integer;          { number to count bits in }
                    var Ones: integer;   { number of "one" bits }
                    var First: integer   { highest "one" bit });
  external;

{ This is a procedure that counts the "one" bits in an integer and
  returns the number of ones in "Ones" and the highest bit found
  in "First".  If no bits are set, "First" receives "-1".
  The procedure uses an extension of Pascal-2 that allows the
  signed number "N" to be treated as an unsigned number "TN".
}
procedure CountOnes;

  var
    TN: 0..65535;                { local unsigned value of N }
    Bits: 0..16;                 { bit count }

  begin
    First := -1;
    Ones := 0;
    Bits := 0;
    TN := N;
    while TN <> 0 do begin
      if odd(TN) then begin
        Ones := Ones + 1;
        First := Bits;
        end;
      Bits := Bits + 1;
      TN := TN div 2;
      end;
  end;
```

This simple procedure counts the number of bits set in an integer, checking whether the lowest bit is set, incrementing a counter, and terminating when there are no more bits set. The use of unsigned integers (TN, in the range 0..65535), avoids the shifting of the sign bit into the lower-order bits. (Unsigned integers are discussed in the Programmer's Guide and in the Language Specification.) This procedure (and many others that are often coded in low-level code) can be coded as a Pascal–2 procedure. But in many ways this procedure is typical of the sort of procedure you may code in MACRO-11:

- It performs a single function with simple internal logic.
- It is a generally useful form of the function, rather than a special use.
- It makes no reference to global variables. All data is passed as parameters.

The first example gives the most direct translation into MACRO-11, with all references to variables made directly to memory. It is quite possible to do the entire function in registers, with some saving

in code and execution time, but for the sake of the example we will not do this. We change the algorithm slightly to make use of the state of the condition code at the end of the loop. The use of a conditional branch at this point shortens execution time slightly at no cost in code size.

```
.title  count

; This is a sample procedure that counts the number of bits
; set to one in a word "n" and also sets the variable "first"
; to the bit number of the highest bit set.
;
; This is used strictly as an example; some values that would
; normally be kept in registers are being kept in local variables
; or handled directly in memory for demonstration purposes.

        proc    countones       ; procedure countones(
        param   n,integer       ;   n: integer;
        param   ones,address    ;   var ones: integer;
        param   first,address   ;   var first: integer);

                                ; var
        var     bits,integer    ;  bits: integer; bit counter

        begin                   ; begin
        mov     #-1,@first(sp)  ; first := -1;
        clr     @ones(sp)       ; ones := 0;
        clr     bits(sp)        ; bits := 0;

        tst     n(sp)           ; if n <> 0 then
        beq     10$             ;
1$:                             ;  repeat
        bit     #1,n(sp)        ;
        beq     2$              ;   if odd(n) then begin
        inc     @ones(sp)       ;    ones := ones+1;
        mov     bits(sp),@first(sp)
                                ;    first := bits;
2$:                             ;    end;
        inc     bits(sp)        ;   bits := bits + 1;
        clc                     ;
        ror     n(sp)           ;   n := n div 2;
        bne     1$              ;  until r0 = 0;
10$:                            ;
        endpr
        .end
```

This procedure illustrates the use of parameters, local variables, and the **begin** and **endpr** macros. The local variable to hold n is not needed as there is no distinction made between signed and unsigned integers at the MACRO-11 level. The equivalent Pascal–2 code in the comments should make the MACRO code easy to follow.

In actual practice, local variables would be kept in registers, and the **save** and **restore** macros would be used to save and restore the registers used. The following version is an example of this

kind of code.

```
        .title  count

        ; This simple procedure counts the number of bits
        ; set to one in a word "n" and also sets the variable "first"
        ; to the bit number of the highest bit set.
        ;
        ; Functionally, this is the same procedure as "examp", except that
        ; it places local variables in registers whenever possible.
        ;
                proc    countones       ; procedure countones(
                param   n,integer       ;   n: integer;
                param   ones,address    ;   var ones: integer;
                param   first,address   ;   var first: integer);

                save    <r0,r1,r2,r3>

                begin
                mov     n(sp),r0        ; r0 := n;
                mov     #-1,r1          ; r1 := -1; first
                clr     r2              ; r2 := 0; ones
                clr     r3              ; r3 := 0; bits

                tst     r0              ; if r0 <> 0 then
                beq     10$             ;
        1$:                             ;  repeat
                bit     #1,r0
                beq     2$              ;   if odd(r0) then begin
                inc     r2              ;    r2 := r2+1;
                mov     r3,r1           ;    r1 := r3;
        2$:                             ;   end;
                inc     r3              ;   r3 := r3 + 1;
                clc                     ;
                ror     r0              ;   r0 := r0 shift 1;
                bne     1$              ;  until r0 = 0;
        10$:                            ;
                mov     r1,@first(sp)   ; first := r1;
                mov     r2,@ones(sp)    ; ones := r2;
                endpr
                .end
```

In the Pascal program that invokes **CountOnes**, the following reference is made:

```
procedure CountOnes(N: integer;
                    var Ones: integer;
                    var First: integer);
  external;
```

Placing PASMAC into the System Macro Library

If you often write Pascal programs that invoke MACRO-11 subroutines written using the PASMAC macro package, you might find it desirable to add the PASMAC package to your system macro library. This allows MACRO-11 programs to use PASMAC via the .MCALL assembler directive rather than by specifying PASMAC.MAC as a separate input file in the command line.

When you assemble a MACRO-11 subroutine, the assembler searches the system macro library (SY:SYSMAC.SML) to define macros requested with the .MCALL directive. The system macro library normally contains definitions of macros that call system services. You can easily add your own macro definitions to this library.

To add PASMAC to the system macro library, perform these steps:

1. Make a backup copy of your system macro library in case something goes wrong.
2. Using a text editor, create a file called P.MAC, which encloses PASMAC.MAC with a macro definition, as shown below. The definition creates a macro called PASMAC, which contains the entire contents of the file PASMAC.MAC. The macro definition redefines the macro PASMAC to be a null macro. This saves space and time in the assembler. The macro definition must be in upper case.

```
.MACRO  PASMAC
   :    ------------------------------ contents of PASMAC.MAC
.MACRO  PASMAC
.ENDM

.ENDM   PASMAC
```

   We offer two ways to create P.MAC. The first way is to create the skeleton macro, then insert the contents of PASMAC.MAC at the location shown above. The other possibility is to first copy PASMAC.MAC to P.MAC and then edit P.MAC, placing the skeleton macro directives around the contents of PASMAC.MAC.

3. Copy SYSMAC.SML to some other file, say SYSMAC.XXX.
4. Run the librarian to add the contents of P.MAC to SYSMAC.XXX. This produces SYSMAC.SML, the new system macro library.

```
.R LIBR
*SYSMAC.SML/M=SYSMAC.XXX,P.MAC
```

PASMAC.MAC is now a part of the system macro library. You can now use the .MCALL directive to define the PASMAC macros in your MACRO routines. This is illustrated below. The routine simply clears registers R0 and R1.

```
.title  test
.mcall  pasmac              ; Read the PASMAC macro
pasmac                      ; Define PASMAC macro

proc    test                ; Sample procedure
param   foo,integer
save    <r0,r1>
begin
clr     r0
clr     r1
endpr
.end
```

Now you can assemble the routine without specifying the PASMAC.MAC source file on the MACRO command line.

```
.MACRO TEST=TEST
```

PROSE: A Text Formatter

Computerized text-processing tools such as text editors and formatters can ease the tedious preparation and editing of computer-oriented documentation. Instead of cutting, pasting, and retyping hard copy, you instruct the computer to insert changes, reformat and number pages, then reprint the document. PROSE, a text-formatting utility program, allows you to print any document in a variety of formats.

This guide describes the operation of PROSE, providing an overview of text-formatting procedures for PROSE and a detailed explanation of PROSE directives. The first-time user of PROSE can read this guide from beginning to end, using it as a tutorial while producing a first document. The more experienced reader may use it as a reference source. As an aid to both, this guide groups PROSE directives by function into four sections: controlling input, establishing format, indexing, and printing. The order of the directives in the guide reflects the order in which these directives might be applied to a text. The reader should have some basic knowledge of a text editor.

PROSE requires a small number of easily learned commands. Unlike some text-formatting programs, which use macro commands, variables, and other features usually associated with programming languages, PROSE does not overwhelm the user with complicated syntax. The text stands out, not the directives. This simplicity allows you to produce high-quality text with a minimum of effort.

Like many text formatters, PROSE will format text in pages, filling and justifying lines, placing titles and page numbers as needed. The following table shows some common features of text formatters that PROSE does and does not have.

Prose Can ...	**... And Cannot**
Underline	Control photo-typesetting machines
Hyphenate words	Do graphics
Convert upper-case input to mixed-case output	Produce multi-column text
Produce a sorted index	Store text and retrieve it later
Print selected pages	Use tabs

PROSE may or may not be the tool for a given application.

PROSE Basics

The basic units of any text-formatting system are the word, the line, and the paragraph. In PROSE, a **word** is defined as any non-blank string of characters, with a blank on either side. For the purposes of formatting, a punctuation character is part of the word next to it. A **line** consists of the number of words that PROSE can fill between margins. PROSE places as many words as possible into each output line, adding blanks to justify the lines to left and right margins.

Text formatting is largely **filling** and **justifying**, a process illustrated by the following example.*

Input to PROSE:

```
  Eddie went to the ground floor cafeteria and got a sandwich
and container of coffee, then went back to his office to work on
the water bill survey. No one else was there; the others were still
out on their regular lunch hour.
  Why not? he asked himself. It took only ten minutes from start to
finish: eight to find the code, one to decide how to do it, and one
more to type the orders into the computer console. When he finished,
the screen told him that the violation number had been removed.
  A few minutes later his office door opened. It was his boss. He was
back ten minutes early from his lunch hour.
  "You're here," the boss said.
  "That's right," Eddie said.
  "Good," the boss said, leaving without bothering to close the
door.
```

Output from PROSE:

```
  Eddie went to the ground floor cafeteria and got a sandwich and
container of coffee, then went back to his office to work on the water
bill survey. No one else was there; the others were still out on
their regular lunch hour.
  Why not? he asked himself. It took only ten minutes from start to
finish: eight to find the code, one to decide how to do it, and one
more to type the orders into the computer console. When he finished,
the screen told him that the violation number had been removed.
  A few minutes later his office door opened. It was his boss. He
was back ten minutes early from his lunch hour.
  "You're here," the boss said.
  "That's right," Eddie said.
  "Good," the boss said, leaving without bothering to close the door.
```

When the user gives no special instructions, called **directives**, PROSE operates in the default mode as shown in the example above. In the default mode, PROSE automatically fills and justifies output lines, formatting the output into pages. Directives instruct PROSE to do anything more sophisticated. If the user doesn't enter certain directives, PROSE supplies their default forms.

---

* Text examples in this guide have been excerpted from *The Programmer* by Bruce Jackson.

Structure of Directive Lines

In general, a directive line has three components: the escape character, the directive name, and the parameter for its application. Most PROSE directive lines take one of three forms:

.directive name
.directive name integer
.directive name(parameter)

The **directive escape character** is placed in the first column of an input line to indicate that at least one directive follows. The period (.) is the default escape character because it seems unlikely that anyone would want to type a period in the first column of a line of text. The default can be changed with the INPUT directive (see "Controlling Input to PROSE").

After the escape character comes the name of the directive that PROSE is to execute. The *directive name* can be abbreviated to three letters (in fact, PROSE only examines the first three). Examples in this guide show directives typed in upper case, but PROSE accepts both cases.

The directive name may or may not be followed by a *parameter*. The BREAK directive, for example, doesn't require a parameter. If a necessary parameter is omitted, PROSE supplies a default value for that parameter. The default values that PROSE uses are listed in a table of options under each directive.

A parameter can be one of three types:

- Text on the remainder of the directive line.
- An integer.
- Any specific options enclosed in parentheses, consisting of other directive names, integers, or keywords defined by the directive itself.

In text-processing systems such as PROSE, a **keyword** (also called a descriptor) categorizes or indexes information. Many PROSE directives use special letters or characters to express the options assigned, as do the INPUT or FORM directives. In directives such as RESET(MARGIN), the keyword is the name of the directive to be changed. The summary directive table in Appendix B indicates the parameter type that each directive may take.

Numeric values used as a parameter or part of a keyword may be either an explicit positive integer or a relative value. A relative value, specified by a plus or minus sign before the integer, indicates that the old value should be increased or decreased by the amount of the integer. For example, if the left margin is set to 10 and the right margin to 70, you could use a relative values in the directive

```
.MARGIN( L+5 R-5 )
```

to squeeze the margins together by 5 characters on each side.

Placement of Directives

Directive lines are usually separated from lines of text (see the following sample file). Several directives can be typed on the same line, provided that they are separated by the directive escape character, as follows.

```
.BREAK.SKIP 2.MARGIN( L5 R65 )
```

Some directives take the remainder of the line as their parameter, so no other directives can follow these (e.g., COMMENT directive). The following sample shows the placement of PROSE commands in an input file.

Input to PROSE:

```
.COMMENT  This example makes very primitive use of directives,
.COMMENT  but it will produce exemplary text.
.MARGIN( L10 R60)
.INDENT 2
Eddie ordered that the tax roll and Yellow Pages tapes be returned
to storage. A few seconds later the video screen told him they had
been returned to their appropriate storage locations.
.BREAK.INDENT 2
Eddie smiled at the screen. He loved the computers. They would do
exactly what you told them to do and they would never lie to you.
Two inhuman characteristics. People who complained about the inhumanity
of computers were right. They didn't know how to care or betray.
.BREAK.INDENT 2
The next operation was more complicated. He had prepared for
it some time before, and the preparation had required many
separate inquiries.
```

Output from PROSE:

```
    Eddie ordered that the tax roll and Yellow Pages
  tapes be returned to storage.  A few seconds later
  the video screen told him they had  been  returned
  to their appropriate storage locations.
    Eddie  smiled  at  the  screen.  He  loved  the
  computers.  They would do exactly  what  you  told
  them  to  do and they would never lie to you.  Two
  inhuman characteristics.  People  who  complained
  about  the  inhumanity  of  computers  were right.
  They didn't know how to care or betray.
    The next operation was more complicated.  He had
  prepared for it some time before, and the prepara-
  tion had required many separate inquiries.
```

For more sophisticated examples, see "Appendix A: Examples of PROSE Directives in Text."

A long directive may extend beyond one line. A continued line is indicated by a **continuation character**, a plus sign '+' placed in column one. The following example shows suggested placement of the continuation character:

```
.FORM( [ /// L58 // #73 'PAGE' p /// ]
+      [ /// L58 //      'PAGE' p /// ] )
```

Generally, directives are placed at the beginning of the input file or at the point in the text where the directive takes effect. Most directives either control the functions of PROSE or set general format guidelines for the document. These are placed at the beginning of the text and their operations are applied throughout, unless temporary changes are made. The **FORM** directive, for example, establishes page format for the rest of the text, but the number of lines per page can be adjusted by an option of the **PARAGRAPH** directive. Directives that apply only to a particular line, such as **INDENT**, **BREAK**, and **COMMENT** in the sample above, are placed wherever necessary throughout the text.

Running the PROSE Program

No actual formatting takes place until the input file, containing text and directive lines, is submitted to PROSE for processing. You create the input file with your system's text editor. PROSE places the formatted text in an output file for submission to a printer or for display on a terminal screen.

To format the input file, invoke PROSE as shown:

```
.R PROSE
*output-file = input-file
```

input-file:

The PROSE source file(s). Multiple input files are read and concatenated from left to right. The default extension for input files is .PRS.

output-file:

The formatted PROSE file. If the output file and the '=' separator are omitted from the command line, the output file will take the name of the last input file and the default extension .DOC. Default output files are placed in the default directory.

The output file may then be printed. The formatting and printing operations may be merged by means of header files.

Header Files

Certain directives nearly always appear at the head of any input file. If all of your documents use these directives in the same form, you can set up a header file rather than type them in each document's input file. Header files also provide you with an easy way to choose among various forms or output devices.

As a general practice, we recommend that you set up each PROSE text without **OUTPUT** or **FORM** directives. Instead, keep these directives in another file that you will use as the first input file, or "header file." For example, you may wish to create a header file for output to a video terminal and another for output to the line printer.

If your header files are stored on the system device, you would use this command to prepare the document for the terminal (assuming the header file is named VT100.PRS):

```
.R PROSE
*DOCNAM = SY:VT100,DOCNAM
```

and this command to prepare the document for the line printer (assuming the header file is named PRINTR.PRS):

```
.R PROSE
*DOCNAM = SY:PRINTR,DOCNAM
```

PROSE prints the output file according to directives in the header file. See "Page Format" and "Specifying Output Devices" for the functions of the individual directives included in the header file.

Controlling Input to PROSE

The directives in this section control the input to the PROSE program. Generally, they are placed at the beginning of the input file for a document or in a header file to be used for all documents. You can set and change them as needed throughout the text.

INPUT Directive

The **INPUT** directive tells PROSE how to interpret certain control characters in the input file and sets the maximum length for input lines.

The following table summarizes the options for its parameter.

Key Letter	Meaning	Type	Default
B	Explicit blank character	character	nul
H	Hyphenation character	character	nul
C	Case-shift character	character	nul
U	Underline character	character	nul
D	Directive escape character	character	.
W	Input width	number	150
K	Keep	number	next

The options, which can be given in any order, consist of a key letter followed by a value. Unless the user specifies both the key letter and a value, the default value is assigned when PROSE begins processing. A value in the parameter changes **only** when a new value is given. No **INPUT** option uses relative values.

B: The **explicit blank character** indicates a blank that PROSE should treat as if it were a character. With the cross-hatch '#' specified as the explicit blank, the following example shows how two words separated by an explicit blank will never be split from one line to the next. PROSE will never fill blanks between the words to justify a line.

```
.INPUT( B# )
...someone like the imaginary Dr.#Conrad and...
```

H: The **hyphenation character** defines hyphenation points within words. Sometimes a long word will cause many blanks to be inserted to justify the preceding line. PROSE will hyphenate such a word if you have defined the syllable boundaries within it. Of course, not all the syllable boundaries need be specified, only those at which you want PROSE to be able to split a word. For example, if the hyphenation character is the slash '/', you may type "syncopation" as **syn/co/pa/tion**. PROSE will insert a hyphen '-' only when the characters on both sides of the hyphenation point are letters. This restriction allows you to type "hyper-active" as **hyper-/active**, and PROSE will split the word if necessary, without adding a superfluous hyphen. If PROSE is forced to insert blanks beyond a certain threshold set by the **OPTION** directive, PROSE will issue an error message on the line that needs hyphenation characters.

C: To produce mixed-case output from upper-case-only input, you must specify a **case-shift character** in the **INPUT** directive parameter, causing PROSE to automatically shift all upper-case letters to lower case. To preserve certain upper-case letters, such as initial capitals for names and sentences, you can surround the letter or letters with case-shift characters. PROSE shifts to upper case for all characters between case-shift characters. "Stuttering" is another way to designate capitals among upper-case-only input. Since most upper-case letters are at the beginning of a word (following a blank), you use two letters to indicate a single capital. Words that already begin with a double letter will produce a single capitalized letter unless you put two case-shift characters before the word.

Input to Pr:	Output from Pr:
^^LLAMA	llama
^^OOPS	oops
LLLAMA	Llama
OOOPS	Oops

The following example demonstrates both ways of producing mixed-case output from upper-case-only input. The case-shift character is easier to use for long strings, such as example programs, that are to be capitalized. Stuttering is easier when you want to capitalize a single character, such as the first word of a sentence. You can use both methods in the same text as shown below.

Input to PROSE:

```
.INPUT(C^)
HHE HAD EIGHTEEN MINUTES, PLUS THE LOCAL CONNECTION. TTWENTY-ONE
MINUTES. AA WORLD OF TIME ON A COMPUTER. HHE WAS READY WITH HIS
QUESTIONS.
  ^WHAT IS CODE FOR PROGRAM?^
  ^COMPUTER BANDIT.^
  ^WHAT IS KNOWN ABOUT COMPUTER BANDIT?^
  TTHE SCREEN RAPIDLY FILLED WITH THE DATES AND AMOUNTS OF HIS
AAMERICAN EEXPRESS THEFTS, SOME OF HIS RECENT INFORMATION SCANS,
THE REPORTS TO THE NEWSPAPERS IN NNEW YYORK.
```

Output from PROSE:

```
He had eighteen minutes, plus the local connection.  Twenty-one
minutes.  A world of time on a computer.  He was ready with his
questions.
  WHAT IS CODE FOR PROGRAM?
  COMPUTER BANDIT.
  WHAT IS KNOWN ABOUT COMPUTER BANDIT?
  The screen rapidly filled with the dates and amounts of his American
Express  thefts,  some of his recent information scans, the reports to
the newspapers in New York.
```

For conversion of mixed-case input to upper-case-only output, see the OPTION directive.

U: Text surrounded by the **underline character** will be underlined. Blanks are not underlined, but explicit blanks are.

D: The **directive escape character** is placed in the first column of an input line to flag it as a directive. Use this option only to define a directive escape character other than the period.

W: The input width W specifies the number of characters to be read from each input line. Users will only need to change the input width for special jobs.

K: The **keep** option explicitly specifies the keep buffer to be used to store the new input options. By default, PROSE uses the numerically next buffer. (See "Changing The Format Control Directives" for detailed explanation of the use of **keep buffers** in PROSE directives.)

OPTION Directive

The OPTION directive gathers together miscellaneous options that affect the filling and justifying PROSE does during text formatting. These options are summarized in the following table. Key letters are followed by a switch symbol (+/-) or an integer, as shown in the default column. For the switch-type options, the plus sign '+' means on and the minus sign '-' means off.

Key Letter	Meaning	Default
E	Print error messages	+
J	Justification limit	3
F	Fill output lines	+
L	Left justify	+
R	Right justify	+
S	Spacing	1
M	Multiple blanks	+
P	Two blanks after periods	+
U	Shift to upper case	-
K	Keep	next

As processing begins, PROSE assigns the default value for each option without a specified value. A parameter value changes only when a specification is given. No option uses relative values.

E: Error messages appear in the formatted text of the main output files at the approximate location of the errors. Error messages are suppressed when this option is off (E-).

J: PROSE inserts blanks as needed to justify the left and right margins of an output line. The **justification limit** controls the point at which PROSE will attempt to hyphenate a word. If, for instance, the justification limit is set at 3, then the hyphenation process will be invoked when PROSE has to insert three blanks between adjacent words on a line. If hyphenation is not possible, or PROSE is not able to bring the number of inserted blanks below the limit, an error message is printed for the line(s).

NOTE

Settings for options E and J can be varied according to the draft you are working on. Setting J to an arbitrarily high number (e.g., 20) and turning off E helps you to avoid hyphenation errors until you are ready to deal with them, usually in the later stages of document preparation.

F: Output lines are automatically filled and justified as described in the "PROSE Basics" section. If the **fill** option is off, PROSE will print the input lines as they are, without reformatting to fill the output lines. In effect, a justification break is done after each input line. Option F- is most useful for literal text, such as program examples, where spacing between words must be exactly as typed.

L: R: The **left** and **right justify** switches work together to determine the justification to be done. If both options are on, output lines are justified to both the left and right margins. If both options are off, the lines are centered between the two margins. If one is on and the other is off, one margin (either left or right) will be straight and the other ragged. The following examples demonstrate the output from the four combinations.

Output from .OPTION(L+ R+) :

```
Eddie did four more operations, three of  them  involving  county  and
city payroll checks, all of which were handled by the same computer.
```

Output from .OPTION(L- R-) :

```
 He gave an across-the-board raise of fifty dollars per check to all
teachers in the ghetto schools.  He deducted an equivalent amount from
     the checks of the city's highly paid political appointees.
```

Output from .OPTION(L+ R-) :

```
Then he erased the tapes for outstanding private residential water
bills.  People, he decided, shouldn't have to pay for a drink of water
or to be able to flush ... a toilet.
```

Output from .OPTION(L- R+) :

```
The final operation was one he hadn't thought of earlier; it had come
        to him during the night's work.  It was easy enough with the
                                                information he now had.
```

S: The **spacing** option 2 generates double-spaced output; the spacing option 3 generates triple-spaced output. By default, text is single-spaced.

M: If the **multiple blanks** option is on (**M+**), multiple blanks in the input file are considered to be significant. That is, if several blanks are placed between two words in the input file, at least that many will appear in the output file; PROSE may add blanks during justification. If the option is off, multiple blanks will be treated as a single blank.

P: The **2 blanks after periods** option places at least two blanks after every period. PROSE will not add blanks before justifying if two are already present. This makes for consistent spacing in the final copy even if you are not careful about typing 2 spaces after sentence periods in the original. Three or more blanks after a period are treated as multiple blanks.

U: For output devices that cannot process mixed-case files, the **shift to upper case** option shifts all lower-case letters to upper-case letters. This option is also useful for printing an entire passage or example, such as a sample program, all in upper case. For conversion of upper-case-only input to mixed-case output, see the **INPUT** directive.

K: The **keep** option explicitly specifies the keep buffer to be used to store the new options. By default, PROSE uses the numerically next buffer. (See "Changing Format Within the Text" for use.)

Setting Up the Document's Format

This section explains the use of directives to format text. These directives specify page format, margins, paragraphing conventions, justification breaks, and blank or comment lines.

Page Format

The **FORM** directive defines the page format, including insertion of titles, date/time, blank lines, page numbers, and other textual items at the top or bottom of the page. The **FORM** directive works with the **COUNT**, **TITLE**, and **SUBTITLE** directives. The **PAGE** and **PARAGRAPH** directives can override the page-break function of the **FORM** directive.

FORM Directive

The **FORM** directive can produce a variety of page formats, depending upon the options specified in its parameter. The table below contains the available **FORM** options; the following paragaphs explain their use.

Key Char	Meaning	Default Field Width
[	Define top of page	–none–
]	Define bottom of page	–none–
#*n*	Tab forward or backward to absolute column *n*	–none–
S	Subtitle	its length
T	Main title	its length
L*n*	Fill in *n* lines of running text on the page	–none–
/	Print an end of line (by itself, a blank line)	–none–
/*n*	Print *n* ends of lines	–none–
P*f*	Current page number, *f* selects the form:	3
	N or n Arabic numerals (default)	[The field
	L Upper-case letter	width will
	l Lower-case letter	be expanded
	R Upper-case Roman numerals	if needed]
	r Lower-case Roman numerals	
'...'	Print material within quotation marks as literal text	–none–
C	24-hour clock as **hh.mm.ss** (e.g. **15.37.58**)	8
D	Raw date as **yy/mm/dd** (e.g. **82/02/13**)	8
E	Nice date as **dd Mmm yy** (e.g. **13 Feb 82**)	9
W	Wall clock as **hh:mm PM** or **hh:mm AM** (e.g. **3:37 AM**)	8

If the **FORM** directive is omitted completely, PROSE uses the default form:

```
.FORM( [ // T #62 E /// L54 /// #33 '- ' PN:1 ' -' //// ] )
```

The sequence of options within parentheses corresponds to the format of the page from top to bottom. The **FORM** directive builds text lines from left to right, starting in the first printable column unless a tabbing specification (#*n*) starts text at a specific column.

FORM directive parameters generally begin and end with the definition characters for a top-of-page and bottom-of-page. The top-of-page definition '[' has several uses. You can direct PROSE to send a page eject to the output device when it reaches the top of a page. Also, you can request a pause at the top of each page to allow you to change paper on the printer (see information on the **OUTPUT** directive in "Printing the Document"). At the end of the document, PROSE signals one last page eject and continues to interpret the **FORM** specification until it reaches another top-of-page. This ensures the execution of any commands specified for the bottom of the last page, such as a page

number. PROSE increments the page number at the bottom-of-page character ']'. So, if you print the page number both before and after the bottom-of-page definition, you will get different numbers.

To print slightly differing formats for facing pages, specify a format for each page between a pair of page definition characters. For example, this form directive prints the page number at the bottom right of odd numbered pages and at the bottom left of even pages.

```
.FORM ( [ // T #62 E /// L56 // #63 'PAGE' P /// ]
+       [ // T #62 E /// L56 //     'PAGE' P /// ] )
```

Appendix A contains another example of the **FORM** directive used to print facing pages.

Page length is determined by the specification for the number of lines. PROSE breaks pages at the number of lines set by the **FORM** directive's L*n* specification, unless the **PAGE** directive or the optional automatic page eject for the **PARAGRAPH** directive is used (see "Page Breaks"). If the L*n* specification is omitted entirely, PROSE supplies the default value of 54 lines per page. If the **FORM** directive parameter contains a key letter **L** without a value, no special page formatting is done. Page length will be infinite, which is useful for working with documents on terminals, where pages are irrelevant. In this mode, a **PAGE** directive with no parameter will put 5 blank lines between sections of text.

Titles, subtitles, page numbers, and dates are placed in fields at the top or bottom of the page. Although default values are sufficent for most situations, the field width can be set to a particular value by placing a colon and the value after the key letter. For example, **T:30** prints the title in a field of 30 characters. Specified field widths are sometimes useful for truncating long titles. PROSE fills the field from right to left. The tabbing specification **#***n* places the field horizontally on the page.

The **FORM** argument is re-scanned as each page of output is produced, so that any change in a title buffer made with the **TITLE** or **SUBTITLE** directive will insert the new title or subtitle on the next page. The **TITLE** directive enters the remainder of the line into the main title buffer. The **FORM** directive uses the contents of the title buffer to print a title on the page as specified. The **SUBTITLE** directive enters the remainder of the line into the subtitle buffer, to be used by the **FORM** directive to print a subtitle on the page as specified. See examples in Appendix A.

PROSE adds a blank line for each '/' mark in the **FORM** directive. These blanks are placed uniformly on each page, in the relative position that they appear in the directive, usually at the top and bottom, between the body of the text and the title and page number. The alternative **/***n* allows a shorter expression of numerous blank lines; either form may be used.

Page numbers are incremented by the page counter and placed on the page by the P*f* option of the **FORM** directive. The **COUNT** directive sets the page counter. The integer in the **COUNT** parameter can be a relative value; for example, **.COUNT +1** increments the page number by one. By default, the page counter sets the page number to 1.

The **literal text** option allows you to add such touches as hyphens surrounding the page number (see default **FORM** directive) or other text that must appear exactly as typed. For example, suppose a press release required the word "more" with parentheses around it at the bottom of each page. You could use the literal text specification as follows:

```
.FORM( [  T /// L54 /// #28 '(more)' /// ] )
```

PROSE users may choose between four styles of dates. The date is placed on the page in much the same manner as a title.

Page Breaks

The **PAGE** directive signals a page eject when fewer than the specified number of lines remain on the current page. If no parameter is given, the **PAGE** directive does an unconditional page eject. The **PARAGRAPH** directive's **automatic page eject** includes the page-break function in the paragraph format for the document. **.PAGE 3** and **.PAR(P3)** are equivalent, except that **.PAGE 3** must be

PARAGRAPH directive's **automatic page eject** includes the page-break function in the paragraph format for the document. .PAGE 3 and .PAR(P3) are equivalent, except that .PAGE 3 must be explicitly placed by the user, while PROSE executes .PARAGRAPH (P3) wherever indicated by the paragraph flag character.

Margins

The MARGIN directive sets the left and right margins for filling and justifying. The value for **left margin** indicates the column in which the line of text begins; the **right margin** value is the column number of the last printed character. Thus, subtracting the left margin from the right margin gives the number of columns for printed text.

The options, which may be given in any order, consist of a key letter followed by a value. The next table lists the key letter for each option.

Key Letter	Meaning	Type	Default	Relative
L	Left margin	integer	0	yes
R	Right margin	integer	70	yes
K	Keep	integer	next	no

Margins are set before text processing begins. PROSE assigns default values of L0 R70 if no MARGIN directive is used or if the directive is given without a parameter. A value changes only when a new specification is given. The **keep** option explicitly specifies the keep buffer to be used to store the new margins. By default, PROSE uses the numerically next buffer.

The INDENT directive moves the next line of text to the right of the page by the given number of spaces. When the directive is used without a parameter, the default value is 5. The UNDENT directive moves the next line a certain number of spaces to the left. (The undent is sometimes known by the name "outdent" or "hanging indent.") If the given parameter would undent the text past the leftmost column of the printed page, the directive undents only to the leftmost printable column. If no parameter is given, the default undents to the leftmost printable column.

Paragraphs

Although you may use any justification break methods to distinguish between one paragraph and the next, the PARAGRAPH directive provides a more versatile method of creating paragraphs.

Placed at the beginning of the text, the directive sets the general form of paragraphs and specifies a **paragraph flag character**. Many PARAGRAPH options take the place of other directives, so the directive is a powerful tool.

When the paragraph flag character is placed in the first column of a text line to signal a new paragraph, PROSE takes any of the following actions that are specified by the parameter.

Key Letter	Meaning	Type	Default
F	Paragraph character	character	nul
I	Automatic indent	number	0
U	Automatic undent	number	0
N	Number generator		-none-
P	Automatic page eject	number	0
S	Automatic skip	number	0
K	Keep	number	next

When it begins processing, PROSE assigns the default value for each option in the **PARAGRAPH** directive parameter. If the input file contains no **PARAGRAPH** directive, or if an option is not specified, the default value is used. No **PARAGRAPH** option uses relative values.

By manipulating the options in the parameter, you may direct PROSE to take any of the following actions for paragraphs.

F: The **paragraph flag character** invokes a collection of paragraphing actions when it appears in the first column of an input line. Note that this character must be set in the first **PARAGRAPH** directive, or no other options apply. As the only specified option, the paragraph flag character signals a justification break.

I:
U: The **automatic indent** or **automatic undent** applies to the first line of the paragraph and moves the line left or right a given number of spaces. If the number generator is used, the indent or undent is applied after the number is generated (see the example using both options below).

N: The **number generator** produces a new number (or letter) for each occurrence of the paragraph flag character. PROSE inserts the number in lieu of the paragraph flag character when the line is formatted, so you must put a space between the paragraph flag character and the text line, if you want one to appear in the output. The number generator is initialized to 1 each time new paragraph settings go into effect. Resumption of an old setting also resumes the old numbering. The number generator's keyword contains these fields (spaces not allowed):

N *numeric-field field-width*

The key characters for *numeric-field* are:

–blank–	No numbering
N or **n**	Arabic numerals
L	Upper-case letter
l	Lower-case letter
R	Upper-case Roman
r	Lower-case Roman

The field width for the numeric field, expressed as an integer, is expanded if necessary. If, for example, you want an Arabic numeral with three spaces left for the numeral, the keyword is **Nn3**.

The next input and output examples illustrate one style of numbered, undented paragraph created with the automatic undent and number generator options. Note that the margin adjustment places the paragraph number in the leftmost column of the printed page.

Input to PROSE:

```
.MARGIN( L10 )
.PARAGRAPH( F& Nn1 U5 )
& Eddie worried about that one for a while, then came up with
a very simple answer:  he would ask the computers if they had
any such self-inspection instructions in their programs.  It
would become part of his regular greeting:  HELLO. HOW ARE YOU
AND ARE YOU PROGRAMMED TO TRAP ME?
```

Output from PROSE:

```
1 Eddie worried about that one for a while, then came up with a
    very simple answer: he would ask the computers if they had
    any such self-inspection instructions in their programs. It
    would become part of his regular greeting: HELLO. HOW ARE
    YOU AND ARE YOU PROGRAMMED TO TRAP ME?
```

P: The **automatic page eject** simulates the effect of the **PAGE** directive. For instance, the directive **.PAR(P4)** causes PROSE to eject a page if fewer than four lines of the paragraph are left at the bottom of the page. The command is applied after the automatic skip.

S: The **automatic skip** functions the same as a **SKIP** directive, placing a blank line before the first line of the paragraph.

K: The **keep** option explicitly specifies the keep buffer to be used to store the new paragraph options. By default, PROSE uses the numerically next buffer.

Values and options can be changed for particular paragraphs or sections of the document, as explained in "Changing Format Within the Text."

Comments

Using the **COMMENT** directive, you can include information in the source of a document that will not be printed in the formatted copy. As shown in the examples in Appendix A, PROSE treats the remainder of the line as a comment and ignores it.

Changing Format Within the Text

To make the fullest use of PROSE, the user must manipulate such options as blank characters, spacing, margins, or page breaks within the text. The **BREAK** and **SKIP** directives allow you to interrupt the established formatting process.

At certain points, you may need to switch formats for specific situations, such as example programs or blocked quotations. **OPTION**, **MARGIN**, and **PARAGRAPH** settings may change frequently, but the number of different settings will probably be predictable and few. Depending upon the number, the variety, and the frequency of changes the text requires, the following techniques can help you to enter new directives or restore previously used options.

Breaking and Skipping Lines

One of the simplest and most frequently used instructions, a **justification break** causes PROSE to stop filling the current output line and print it without justifying. A line break can be indicated in many ways. Text can be separated (broken) by one or more blank lines inserted in the text,

by leading blanks typed on an input line (a paragraph indention), or by the **BREAK** directive. The following example illustrates these three methods.

Input to PROSE:

```
"We've got to feed him an estimate that is believable--"
  "--but totally inaccurate," the IBM man said.
.BREAK
"Right," Barstow said.
.BREAK
"It's like Battleship," Purvey said, smiling at Barstow
and the IBM man.

"You understand perfectly," the IBM man said, "That's
exactly what it is."
```

Output from PROSE:

```
"We've got to feed him an estimate that is believable--"
  "--but totally inaccurate," the IBM man said.
"Right," Barstow said.
"It's like Battleship," Purvey said, smiling at Barstow  and  the  IBM
man.

"You  understand perfectly," the IBM man said, "That's exactly what it
is."
```

With any of these methods, you will only direct PROSE to do a justification break. PROSE will not skip lines or indent unless you explicitly enter blank lines or indentions in the input file.

The **SKIP** directive prints blank lines within the text by skipping a certain number of output lines. **SKIP** will not print blank lines at the top of a page, unless you enter at least one actual blank line before the **SKIP** directive. The default value of the **SKIP** directive is 5 lines.

Keep Buffers

The **keep buffer** is a simple way to change directives that control input or format. Each time a change in one of these directives is processed, PROSE saves the new values in a **keep buffer**. Ten keep buffers (0 through 9) are associated with each directive. You may use a keep parameter to specify the buffer to be used; if no buffer is specified, the values are saved in the numerically next buffer. To recall a previously used value, you enter the directive with the number of the keep buffer as the parameter.

For example, suppose that a double-spaced text has a number of paragraph-length quotations, which are to be typed as single-spaced blocks indented ten spaces from each margin. Using the keep option in the parameters for the **INPUT**, **MARGIN**, and **PARAGRAPH** directives, you could store the format specifications for each situation in two different keep buffers by entering these directives at the beginning of the input file:

```
.OPTION(K1 S2).MARGIN(K1 L10 R70).PARAGRAPH( K1 F& I2 S1 )
```

Then you enter these directives for the new format

```
.OPTION(K2 S1).MARGIN(K2 L20 R60).PARAGRAPH( K2 F& I0 S0 )
```

before the first blocked paragraph. To resume the standard paragraph format, you then enter the directive names with the number of the keep buffer. The input and output files would look like this:

Input to PROSE:

```
.OPTION( K1 S2 ).MARGIN( K1 L10 R70).PARAGRAPH( K1 F& I2 S1 )
&There was nothing to be done.  He had once heard a comedian
say,
.OPTION( K2 S1 ).MARGIN( K2 L20 R60 ).PARAGRAPH( K2 F& I0 S0 )
If you don't like the telephone company,
you know what you can do? Two tin cans and
a piece of string, that's what you can do.
That's the only alternative you've got.
.OPT 1.MAR 1.PAR 1.SKIP 1
The people in the audience had laughed. Eddie thought about
it now and decided it wasn't funny at all.
```

Output from PROSE:

```
        There was nothing to be done.  He had once heard a

comedian say,

            If you don't like the telephone company,
            you know what you can do? Two tin cans
            and a piece of string, that's what you
            can do.  That's the only alternative
            you've got.

The people in the audience had laughed.  Eddie thought about

it now and decided it wasn't funny at all.
```

To change format for the next single-spaced, blocked paragraph, you would only need to enter:

```
.OPT 2.MAR 2.PAR 2
```

The example is a little more cumbersome than is necessary for one format change. Actually, you need only enter .OPT.MAR.PAR to return to keep buffer 1 in the text shown above. When no parameter is specified, the values are set to those stored in the numerically previous keep buffer, since the keep number is automatically incremented whenever a directive is entered and automatically decremented when that directive is entered without a parameter. Used in this way, the keep buffers function as a "stack" for temporary storage of variations from a basic format.

Reset Directive

The RESET directive sets twelve frequently changed directives to their default values. The following table summarizes the effect of the RESET directive on each:

Directive	Effect
INPUT	Default values for all options.
OPTION	Default values for all options.
FORM	Default values for all options, causes page eject.
COUNT	Sets page counter to 1.
TITLE	No titles until reentered.
SUBTITLE	No subtitles until reentered.
PAGE	Causes a page eject.
INDEX	Deletes all accumulated entries.
MARGIN	Default values for all options.
PARAGRAPH	No paragraphing until directive reentered.
OUTPUT	No pause at top of page; carriage return to do underlining; causes page eject.
SELECT	Discontinues page selection; all pages to be printed.

The RESET directive can be used three ways:

- Entering the directive name with no parameter resets the values of all directives to their defaults:

```
.RESET
```

- Using a directive name as a keyword resets the selected directive. For example, this command resets only the MARGIN and OPTION directives:

```
.RESET( MARGIN OPTION )
```

- Stating the keyword "except" and a directive name in the parameter excludes a selected directive. For example, this directive resets all directives with the exception of FORM and OUTPUT:

```
.RESET( EXCEPT FORM OUTPUT )
```

New directives may be entered after the RESET directive. The RESET directive is an easy way to clear a complicated series of format changes from the keep buffers.

Creating An Index

The INDEX and SORTINDEX directives provide the information PROSE needs to create an index for the document. The INDEX directive is entered as .INX to distinguish it from .INDENT and takes the remainder of the line together with the current page number as an index entry. As the formatted text migrates from page to page in various drafts, the page numbers in the index are updated.

Index entries accumulated by INX directives can be sorted alphabetically or by page number, then printed in a relatively flexible manner. The SORTINDEX directive allows you to specify the method of sorting entries and the format for printing the index.

The options for the SORTINDEX directive, listed in the following table, may be given in any order.

Key Letter	Meaning	Default
S	Sorting option. If this is numeric, it is the first significant column for alphabetic sorting. If it is the letter P, sorting is selected by page number.	1
M	Margin (left margin before index line)	0
P	Column (in index entry) to insert page number	0
L	Left width of page number (field width of number)	2
R	Right width of page number, blanks printed after	2

In the absence of a parameter, default values are used.

Printing the Document

This section does not anticipate all the possible idiosyncracies of the PDP-11 and its peripherals; it gives you directions for printing all or part of the document on certain standard devices.

Specifying Output Devices

The OUTPUT directive defines important aspects of the output device to which you will send the formatted text. The directive takes the general form:

.OUTPUT(*device,options...*)

All of the following is specific to the PDP-11.

One of the following acronyms indicates the output device to be used.

ASC ASCII terminals use the backspace for underlining, but are otherwise the same as the lineprinter (LPT) below. Pauses for page eject, however, are handled differently (see the P option below).

LPT Line printers use overprinting with a carriage return to do underlining. This is the default output device.

The following table contains the options for each output device. Keyword type is an integer or a switch.

Key Letter	Meaning	Default
E	Page eject at top of page ([in FORM description)	-
P	Pause at top of page	-
S	Shift output lines to the right	0
U	Underlining is available	+

These options may be given in any order.

E: The **page eject** option prints a form feed character for each time PROSE reads [in the FORM specification.

P: The **pause** option causes PROSE to stop printing and await operator acknowledgement each time a '[' character is encountered in the FORM specification. On an ASC terminal, PROSE will sound the bell and wait for a carriage return to be entered. For an LPT output device, no action will be taken.

S: The **shift** option shifts all PROSE output to the right by any number of spaces up to 50. This makes it easy to center output on a wide printer page.

U: If the destination terminal does not have underlining capability and the input file contains underline characters, the **underlining available** option should be turned off to prevent PROSE from trying to generate overprinted underlines.

Usually, the OUTPUT directive appears only at the beginning of the input file or in a header file. However, it must also be used immediately after a .RESET(OUTPUT) directive.

The LITERAL directive is useful for producing special printer control characters on some systems. It prints the remainder of the input line as a single output line, after special processing for upper and lower case, underlining, and literal blanks. This single line is printed independently of filling and justifying, or page-formatting processes; it is not counted as an output line.

Printing Selected Pages and Sections

The SELECT directive will print specified pages of a document. Like the OUTPUT directive, the SELECT directive is placed before any lines that are to be printed on the output device, perhaps in a header file. Although the entire text will be formatted, only the selected pages will be printed, saving unnecessary printing time.

The parameter consists of page numbers separated by spaces. Two page numbers separated by a colon will select the span of pages, including beginning and ending numbers. As shown below, the plus sign '+' specifies a second page number relative to the first. The following example prints pages 3, 5, 10 through 15 inclusive, and 20 through 25 inclusive.

```
.SELECT( 3 5 10:15 20:+5 )
```

By default, all pages are printed.

Appendix A: Examples of PROSE Directives in Text

Example 1.

Input to PROSE:

```
.OUTPUT (LPT U+ E+)
.COMMENT  +-------------------------------------------------+
.COMMENT  |NOTE:  The header file supplied by Oregon Software  |
.COMMENT  |contains the output command for the line printer.   |
.COMMENT  |Do not use the above command if using that header.  |
.COMMENT  +-------------------------------------------------+
.FORM ([ // #10 Pn #28 T /// L50 //// ]
+      [ // #60 Pn #17 S /// L50 //// ])
.COMMENT  Page numbering starts at 62
.COUNT 62
.COMMENT  This combination of form and count directives
.COMMENT  duplicates the facing-pages format used in the
.COMMENT  many typeset books.
.COMMENT
.TITLE The Programmer
.SUBTITLE Chapter Three
.MARGIN(L10 R62)
.INPUT (H/ U_)
.PARAGRAPH (F& I2)
&Computers, Eddie knew, have no idea where their sources of
information are in the world. They look upon the world as one great
big fat wire bulging with information and instructions, a wire with
no beginning, middle, or end. The world for a computer is merely an
electrical input saying, ''Here's what you should know,'' or, ''Here's
what I want to know,'' or, ''Here's what you are to do now,'' and an
electrical output for them to talk back:
''Here's what I must know to answer your question,'' and, ''Here are
your answers.''  To the computers, all interrogators and commanders
speak with the same voice and the same authority; all listeners have
the same ear.
&Like guns. It doesn't matter to a gun who pulls its trigger. Guns
have awesome power, but they are entirely dependent on the hands that
use them. Morally guns and computers are out of it all, though they
are regularly the instruments for people who make things happen.
&Twice now--first with Betty's parking ticket and now with his
$25,624.34--Eddie had been someone who made things happen. It was
a very exciting sen/sa/tion, one he hadn't previously experienced.
&He had seen, not long before on the ''Today'' show, an airline pilot
who talked about his af/fec/tion for the 747. He loved it more, he said,
than any other aircraft he had ever flown, and he had been a pilot for
twenty-five years. The inter/view/er asked him to explain his
enthusiasm.
&''I sit there in that little room in the front,'' the pilot said, ''a
room just four stories off the ground when we're parked, but at the
top of the world when we're in flight--and I move litle knobs and
dials. None of them takes more than a few ounces of pressure.
```

```
In an instant a machine weigh/ing a hundred tons responds more smoothly
than if I were moving it myself. It's like the aircraft becomes an
extension of myself because so little effort is needed to make it do
what I want, and it does whatever I want it to do. It's very
ex/cit/ing.''
&''You make it sound almost sexual,'' the inter/view/er said.
&The pilot frowned. ''I don't know about that. I never thought about
that.'' His face brightened. ''It's not sex. It's better. It's _real_
power.''
&Eddie sensed that his entire relationship with the computer had
started a radical change. Before, he had been the machine's servant,
bringing it little orders and loads of information to feed upon. The
questions weren't his and the answers never mattered to him. He was
merely an intermediary in the affairs of others. Now he was having
his own affair.
&And his own affair required further action before the check in his
pocket became anything but a useless piece of paper.
.SKIP 2
&On his way back to the office Eddie stopped in the motor vehicle
section. Edna was there alone, as usual, talking on the telephone,
also as usual. She was wearing a different pink sweater. She held up
her hand, the fingers all extended. At first Eddie thought she was
showing off her rings--there were four of them, all different--but
then he understood that she was telling him she would be on the phone
five minutes longer. He waved his hand to indicate he was in no hurry.
She leaned back in the chair, her pink breasts pointing toward the
corner light fixture.
&Eddie wandered around the office, acting as if he were bored. He
looked at some papers. She was paying no attention to him. He stopped
at the drawer where blank drivers' licenses were kept. He looked over
his shoulder. She was still talking, her back to him. Her left hand
held the telephone and her right hand slowly rubbed the back of her
neck.
&He quickly took from the drawer ten forms, then leaned on the
counter and quietly stamped each of them with the tricolored state
seal required for validation. He put the forms into his jacket pocket
along with the two checks. Now he had only to type out whatever names
he wanted to use and he would have official New York certification.
```

Output from PROSE:

62 The Programmer

Computers, Eddie knew, have no idea where their sources of information are in the world. They look upon the world as one great big fat wire bulging with information and instructions, a wire with no beginning, middle, or end. The world for a computer is merely an electrical input saying, ''Here's what you should know,'' or, ''Here's what I want to know,'' or, ''Here's what you are to do now,'' and an electrical output for them to talk back: ''Here's what I must know to answer your question,'' and, ''Here are your answers.'' To the computers, all interrogators and commanders speak with the same voice and the same authority; all listeners have the same ear.

Like guns. It doesn't matter to a gun who pulls its trigger. Guns have awesome power, but they are entirely dependent on the hands that use them. Morally guns and computers are out of it all, though they are regularly the instruments for people who make things happen.

Twice now--first with Betty's parking ticket and now with his $25,624.34--Eddie had been someone who made things happen. It was a very exciting sensation, one he hadn't previously experienced.

He had seen, not long before on the ''Today'' show, an airline pilot who talked about his affection for the 747. He loved it more, he said, than any other aircraft he had ever flown, and he had been a pilot for twenty-five years. The interviewer asked him to explain his enthusiasm.

''I sit there in that little room in the front,'' the pilot said, ''a room just four stories off the ground when we're parked, but at the top of the world when we're in flight--and I move litle knobs and dials. None of them takes more than a few ounces of pressure. In an instant a machine weighing a hundred tons responds more smoothly than if I were moving it myself. It's like the aircraft becomes an extension of myself because so little effort is needed to make it do what I want, and it does whatever I want it to do. It's very exciting.''

''You make it sound almost sexual,'' the interviewer said.

The pilot frowned. ''I don't know about that. I never thought about that.'' His face brightened. ''It's not sex. It's better. It's real power.''

Eddie sensed that his entire relationship with the computer had started a radical change. Before, he had been the machine's servant, bringing it little

orders and loads of information to feed upon. The questions weren't his and the answers never mattered to him. He was merely an intermediary in the affairs of others. Now he was having his own affair.

And his own affair required further action before the check in his pocket became anything but a useless piece of paper.

On his way back to the office Eddie stopped in the motor vehicle section. Edna was there alone, as usual, talking on the telephone, also as usual. She was wearing a different pink sweater. She held up her hand, the fingers all extended. At first Eddie thought she was showing off her rings--there were four of them, all different--but then he understood that she was telling him she would be on the phone five minutes longer. He waved his hand to indicate he was in no hurry. She leaned back in the chair, her pink breasts pointing toward the corner light fixture.

Eddie wandered around the office, acting as if he were bored. He looked at some papers. She was paying no attention to him. He stopped at the drawer where blank drivers' licenses were kept. He looked over his shoulder. She was still talking, her back to him. Her left hand held the telephone and her right hand slowly rubbed the back of her neck.

He quickly took from the drawer ten forms, then leaned on the counter and quietly stamped each of them with the tricolored state seal required for validation. He put the forms into his jacket pocket along with the two checks. Now he had only to type out whatever names he wanted to use and he would have official New York certification.

Example 2.

Input to PROSE:

```
.COMMENT  Output directive is in the header file.
.INPUT( B# H/ )
.OPTION( K1 )
.FORM( [ // #50 W #60 E /// L50 / #30 '- ' PN:1 ' -' ] )
.MARGIN( K1 L5 R70 )
.PARAGRAPH( K1 F& S1 )
&Something clicked in another part of his mind and he knew he was about
to become a portable computerized superpower.
&The question had been puzzling him for some time. It had to do with
pro/gram access. If one had a pro/gram--if one knew the para/dig/matic
structure of a set of encoded information--then one could do nearly
any/thing one wanted with that information. If it was simply material
stored, then one could learn everything that was stored; if it was
operational material, then one could command the operations. The
problem was, one needed the program to do the work, and the utility
of the programs he had taken with him when he left Buffalo was
limited.
&The cold water swirled around his legs and the ripples moved out from
where his hands paddled the surface. Suddenly it was as if the answers
had typed themselves out on the console screen.
.OPT( K2 U+ S2 )
.MAR( K2 L15 )
.PARAGRAPH( K2 F& U8 S1 )
&QUESTION:##HOW DO I FIND OUT WHAT PROGRAMS EXIST
WHEN I DON'T KNOW WHAT QUESTIONS TO ASK?
&ANSWER:##ASK THE COMPUTERS WHAT QUESTIONS THEY CAN
ANSWER FOR YOU. IF YOU HAVE THE ANSWERS, YOU KNOW THE QUESTIONS.
.OPT.MAR.PAR
&He ran home through the woods without even bothering to
dry off. Mosquitoes pecked at his face. He dressed quickly,
hooked up the van, and sat down at his keyboard. He addressed
IFFI, the central law enforce/ment computer in Baltimore. The
acronym stood for Information#Filed#for#Future#Investigations.
He asked IFFI a question that translated as, ''What discrete
sets of information have you on hand and what are the access codes
for them?'' He set the machine for a printout rather than a
readout on the monitor.
&In seconds the Selectric began typing away. It typed for a long
time. Office Selectrics can handle about thirty characters a
second, faster than any human can go, but the ones built for
information processing went three times as fast. Nearly
one thousand five-/character units of information a minute, and
the machine seemed to be typing faster than he had ever seen it
go before...
```

PROSE: A Text Formatter

Output from PROSE:

3:35 PM 14 Jun 82

Something clicked in another part of his mind and he knew he was about to become a portable computerized superpower.

The question had been puzzling him for some time. It had to do with program access. If one had a program--if one knew the paradigmatic structure of a set of encoded information--then one could do nearly anything one wanted with that information. If it was simply material stored, then one could learn everything that was stored; if it was operational material, then one could command the operations. The problem was, one needed the program to do the work, and the utility of the programs he had taken with him when he left Buffalo was limited.

The cold water swirled around his legs and the ripples moved out from where his hands paddled the surface. Suddenly it was as if the answers had typed themselves out on the console screen.

QUESTION: HOW DO I FIND OUT WHAT PROGRAMS EXIST WHEN I DON'T KNOW WHAT QUESTIONS TO ASK?

ANSWER: ASK THE COMPUTERS WHAT QUESTIONS THEY CAN ANSWER FOR YOU. IF YOU HAVE THE ANSWERS, YOU KNOW THE QUESTIONS.

He ran home through the woods without even bothering to dry off. Mosquitoes pecked at his face. He dressed quickly, hooked up the van, and sat down at his keyboard. He addressed IFFI, the central law enforcement computer in Baltimore. The acronym stood for Information Filed for Future Investigations. He asked IFFI a question that translated as, ''What discrete sets of information have you on hand and what are the access codes for them?'' He set the machine for a printout rather than a readout on the monitor.

In seconds the Selectric began typing away. It typed for a long time. Office Selectrics can handle about thirty characters a second, faster than any human can go, but the ones built for information processing went three times as fast. Nearly one thousand five-character units of information a minute, and the machine seemed to be typing faster than he had ever seen it go before...

Appendix B: Summary Directive Table

Directive	Meaning (action)	Break	Parameter Type	Default
BREAK	Break justification	*	–none–	–none–
COMMENT	No action		remainder of line	–none–
COUNT	Set page count		numeric	.COU 1
FORM	Define page format	*	(...)	.FOR([/2 T #62 E /3 L54 + /3 #33 '-' PN:1 '-' /4])
INDENT	Indent following line	*	numeric	–none–
INPUT	Specify input options	*	(...) or numeric	.INP(D.W150 K+1)
INX	Store index entry		remainder of line	–none–
LITERAL	Print literal text		remainder of line	–none–
MARGIN	Set margins	*	(...) or numeric	.MAR(L0 R70)
OPTION	Set options	*	(...) or numeric	.OPT(S1 F+ M+ P+ L+ + R+ J3 E+ U- K+1)
OUTPUT	Specify output device		(...)	.OUT(LPT, S0 U+)
PAGE	Eject to top of page	*	numeric	.PAG 5
PARAGRAPH	Set paragraphing params		(...) or numeric	–none–
RESET	Reset directive defaults	*	(...)	–none–
SELECT	Select pages to print	*	(...)	–none–
SKIP	Skip output lines	*	numeric	–none–
SORTINDEX	Sort and print index	*	(...)	.SOR(S1 L2 R2)
SUBTITLE	Set the subtitle		remainder of line	–none–
TITLE	Set the main title		remainder of line	–none–
UNDENT	Undent following line	*	numeric	–none–

The directives marked with an asterisk (*) cause a justification break before they are processed, since they affect the filling and justifying environment.

The ellipsis (...) indicates that the parameter is enclosed in parentheses and is described in detail along with the description of the directive itself.

Appendix C: Historical Notes

Most text-formatting programs available today descend from one of several original programs. Among these is RUNOFF, developed on the Dartmouth Time-Sharing System in the 1960s. Later, the Call-a-Computer system provided a RUNOFF version called EDIT RUNOFF as a text-editor command. In 1972, Michael Huck, working on the University of Minnesota's MERITSS system (a CDC 6400 running the KRONOS operating system), began to develop a version of EDIT RUNOFF called TYPESET.

TYPESET was intended as a "versatile text information processor commonly used to typeset theme papers, term papers, essays, letters, reports, external documentation . . . , and almost any other typewritten text."* In spite of these aspirations, no program can be all things to all people. TYPESET went through many changes, stablizing somewhat in early 1977 at version 5.0, which is written in CDC COMPASS assembly language. John P. Strait developed PROSE, written in Pascal, over a year's time starting in the spring of 1977. The design was influenced heavily by TYPESET, making PROSE one of the many descendants of RUNOFF.

PROSE, with minor changes, was installed on the Univac 1100 series computers in early 1980 by Michael S. Ball of the Naval Ocean Systems Center. At Oregon Software, he converted this version from the Univac to the PDP-11 in July 1980 and to the Motorola MC68000 in the spring of 1982.

---

* Michael Huck, *Typeset 5.0 Information*, © 1977.